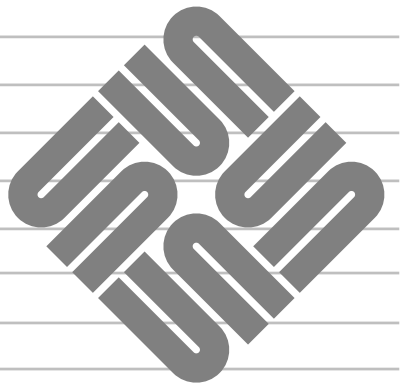
sun microsystems

# Writing FCode Programs for SBus Cards



Part No: 800-4456-10
Revision A of July, 1990

Products Rights Notice:



You understand that these materials were not prepared for public release and you assume all risks in using these materials. These risks include, but are not limited to errors, inaccuracies, incompleteness and the possibility that these materials infringe or misappropriate the intellectual property right of others. You agree to assume all such risks.

THESE MATERIALS ARE PROVIDED BY THE COPYRIGHT HOLDERS AND OTHER CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS (INCLUDING ANY OF OWNER'S PARTNERS, VENDORS AND LICENSORS) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THESE MATERIALS, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Sun, Sun Microsystems, the Sun logo, Solaris, OpenSPARC T1, OpenSPARC T2 and UltraSPARC are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd. The Adobe logo is a registered trademark of Adobe Systems, Incorporated. Part of the products covered by these materials may be derived from the Berkeley BSD systems licensed by the University of California. Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product described in these materials. This distribution may include materials developed by third parties who have intellectual property rights therein. Products covered by and information contained in these materials may be controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists may be prohibited.

# Contents

## Writing FCode Drivers for SBus Cards

# *About This Book*

This manual, *Writing FCode Drivers for SBus Cards,* describes how to write FCode device drivers for SBus cards, how to test those drivers, and how to burn the tested drivers into PROMs.

## Who Should Read this Book

This manual is written for designers of SBus interface cards. It assumes that you have some familiarity with SBus card design requirements and Forth programming.

This manual also assumes that you have read and understood the *SBus Specification*, *Open Boot PROM Toolkit User's Guide*, and *Writing SBus Device Drivers*.

## Scope of This Book

This manual covers the following topics:

- ❏ Describing SBus card testing tools
- ❏ Comparing FCode and Forth-83
- ❏ Writing and Testing FCode programs
- ❏ Putting tested FCode in PROM

## Typographic Conventions

This manual follows a number of t

- *This font* is used for emphasis, for a command argument, and for the title of a book. For example:

  You must type the *filename* argument as described in the *SunOS Reference Manual*.

- `This font` indicates a program listing, a command name, a program name, or text the machine displays on the screen, as in a tutorial session. For example:

  ```
  You have new mail.
  ```

- **`This font`** indicates what you type during a tutorial session. For example:

  ```
  tutorial% date
  ```

- **This font** also indicates the first definition in the manual of an important concept or function. For example:

  Each item's attributes are saved in a **device tree.**

- A rectangular box around text indicates a key name or a panel button on a window-based program. For example:

  Press the Control-C key.

  When you see two key names within one rectangular box, press and hold the first key down, then press the second key. For example:

  To press Control-C , press and hold Control , then press C .

- In a SunOS™ command line, square brackets indicate an optional entry and italics indicate an argument that you must replace with the appropriate text. For example:

  `cd [`*directory*`]`

## References

This manual does not pretend to c
know to write FCode drivers for SB
some other books, too.

For information about SBus, Open PROM, SBus device drivers, and writing device drivers for Sun workstations, see the following Sun manuals:

- *Open Boot PROM Toolkit User's Guide*
- *Open Boot PROM Reference Card*
- *SBus Specification*
- *Writing SBus Device Drivers*

For information about Forth and Forth programming, see:

- *Mastering Forth,* Anita Anderson and Martin Tracy. Brady Communication Company, Inc., 1984.
- *Forth: A Text and Reference*, Mahlon G. Kelly and Nicholas Spies. Prentice Hall.
- *Starting FORTH*, Leo Brody. FORTH, Inc., 1981.

## Software Tools

Some programs specifically mentioned in this manual for use in developing FCode programs are included on a diskette in the SBus Developer's Kit. These programs include the tokenizer, `reheader.fth`, `pburn`, and so on. Instructions for using these programs are included on the diskette.

If you don't have access to a complete SBus Developer's Kit, or if your SPARCstation doesn't have a diskette drive, contact the Sun SBus Support Group for the software.

# 1

# *SBus Cards and FCode*

## Introduction

Each SBus card must have a PROM whose contents identify the device.

The SBus card's PROM may also include an optional software driver that lets you use the card as a boot device or a display device during booting. The software driver may also include diagnostic self-test code.

In addition to designing hardware, the process of developing SBus devices may include writing, testing, and installing FCode drivers for the device. These drivers, if present, serve two functions:

- To exercise the device during development, and to verify its functionality.
- To provide the necessary driver to be used by the boot PROM during power-up.

In practice, these two functions overlap substantially. The same code needed by the boot PROM usually serves to significantly test the device as well, although additional code may be desired to fully verify proper behavior of the device.

The PROM code is used before and during the boot sequence. After the boot sequence finishes, and while not using the PROM monitor, most SBus device use is through SunOS drivers.

SBus device PROMs must be written in the FCode programming language, which is similar to Forth-83. FCode is described in more detail in Chapter 2, "Elements of FCode Programming".

## FCode PROM Format

An FCode PROM begins at address `0` within the SBus card's physical address space. Its size can range from 30 bytes up to the entire physical address space of the SBus card. Typical sizes are 60 bytes (for a simple card that identifies itself but does not need a driver) and 1-4K bytes (for a card with a boot driver). It is good practice to make FCode boot drivers as short as is practical.

An FCode PROM must be organized as follows:

- ❏ Header (8 bytes: consisting of magic number, version number, length, checksum).
- ❏ Body (FCode program; 0 or more bytes).
- ❏ End Token (either `End0`, a zero byte, or `End1`, an alternative all 1's byte).

## Interpreting FCode

For each SBus slot, the FCode program is interpreted during bootup as follows:

- ❏ If there is no response (as when there is no card in that slot), the slot is subsequently ignored.
- ❏ Location 0 of the SBus PROM is read with a 32-bit access. The card must either return the first 4 bytes of the PROM, or return the first byte and a byte acknowledgment so that the CPU will perform bus sizing for the remaining 3 bytes.
- ❏ If the high-order byte of the value returned from the first access is not the FCode magic number `0xfd`, the slot is subsequently ignored.

- If the high-order byte is `0xfd`, the PROM is assumed to contain a valid FCode program. The FCode is then interpreted by starting at location `0` and reading one byte at a time, executing a procedure associated with each FCode value.
- When a byte containing either `0x00` or `0xff` (`End0` or `End1`) is encountered, interpretation ceases.

## Device Identification

An FCode PROM must identify its device. This identification must include, at a minimum, the driver name, used to link the device to its SunOS driver. Identification information may include additional characteristics of the device for the benefit of the operating system and the CPU boot PROM.

In most systems, the CPU's FCode interpreter will store each device's identification information in a **device tree** that has a node for each device. Each **device node** has a **property list** that identifies and describes the device. The property list is created as a result of interpreting the program in the FCode PROM.

Each property must have a name and a value. The name is a string and the value is an array of bytes, which may encode strings, numbers, and various other data types.

Properties may be created by FCode PROMs. The CPU boot PROM understands certain property names that tell it things such as the type of the device (disk, tape, network, display, and so on). The CPU boot PROM may use this information to determine how to use the device (if at all) during the boot process.

SunOS understands other property names that give information used for configuring the operating system automatically. These properties include the driver name, the addresses and sizes of the device's registers, and interrupt levels and interrupt vectors used by the device.

Other properties may be used by individual UNIX™ device drivers. The names of such properties and the interpretation of their values is subject to agreement between the writers of the FCode PROM and the UNIX driver, but may otherwise be arbitrarily chosen. For example, a display device might declare width, height, and depth properties to allow a single UNIX

driver to automatically configure itself for one of several similar but different devices.

# 2

# *Elements of FCode Programming*

## Comparing FCode and Forth-83

FCode is based on the Forth-83 dialect of the Forth language, with the following major differences:

- The FCode tokenizer program uses normal textfiles, rather than the BLOCKs and block editing of Forth-83, and contains its own pre-defined words for file transfers.
- Forth-83 is designed for 16-bit machines. FCode is designed for 32-bit machines, so FCode handles 16 and 32-bit quantities differently than Forth-83.

FCode has these characteristics:

- The source format is machine and system independent.
- The binary format (FCode) is machine, system, and position independent.
- The binary format is compact.
- The binary format may be interpreted easily and efficiently.
- Programs are easy to develop and debug.

- ❑ The source format can easily be translated to binary format.
- ❑ The binary format can be untranslated back to a source format.

Forth commands are called **words**, and are roughly analogous to procedures in other languages. Unlike other languages, such as C, which have operators and syntactic characters and procedures, in Forth every word is a procedure.

Forth words consist of one or more printable characters, separated by one or more spaces. If you define a new word having the same spelling as an existing word, the new definition supersedes the older one(s).

Forth uses a left-to-right reverse Polish notation, like some scientific calculators. The basic structure of Forth is: do this, now do that, now do something else, and so on.

New Forth words are defined as sequences of previously existing words. Subsequently, new words may be used to create still more words.

FCode is a byte-coded translation of a Forth program. Translating Forth source code to FCode involves replacing the Forth word names (stored as text strings) with their equivalent FCode numbers. The tokenized FCode takes up less space in PROM than the original ASCII textfile form of the Forth program from which it was derived.

For purposes of this manual, the term FCode indicates both binary-coded FCode and the Forth programs written as ASCII text files for later conversion to binary-coded FCode.

Except where a distinction between the two forms is explicitly stated, the use of FCode in this manual can be assumed to apply equally to both FCode and Forth.

Two concepts are critical to understanding FCode (or Forth):

- A **colon definition** creates a new word with the same behavior of a sequence of existing words. A colon definition begins with a colon and ends with a semicolon.
- Most parameter passing is done through a push-down, last-in, first-out **stack**.

## Colon Definitions

Normally, the action associated with an FCode word is performed when the FCode word is encountered. This is called **interpret state**. However, you can switch from interpret state to **compile state**.

In interpret state, FCode words are executed as they are encountered. Interpret state operates until encountering a ":". The word ":" does the following:

- Allocates a new FCode word and associates it with the name immediately following the colon
- Switches to compile state

During compile state operation, FCodes are saved for later execution, rather than being executed immediately. The sequence thus compiled is installed in the action tables as a new word, and can be later used in the same way as if it were a built-in word.

Compile state continues until a ";" is read, switching operation back to interpret state.

FCode words encountered after the colon are compiled into RAM for later use, until a semicolon is encountered. The word ";" does the following:

- Compiles an end-of-procedure FCode word
- Switches to interpret state

After compilation, the newly-assigned FCode word can be either interpreted or compiled as part of yet another new word.

Here's an example of a colon definition, defining a new FCode word `dac!`:

```
: dac! ( data addr reg# -- ) swap dac ! dac + ! ;
```

## Stack Operations

Each FCode word is specified by its effect on the stack and any side effects, such as accessing memory. Most FCode words affect only the stack, by removing arguments from the stack, performing some operation on them, and putting the result or results back on the stack.

The stack effects of an FCode word is described by a **stack comment**, included in the colon definition.

In the previous example , the stack comment, beginning with "`(` " and ending with "`)` ", shows that `dac!` takes three parameters from the stack, and doesn't replace them with anything when it's done.

You can place stack comments anywhere in a colon definition, and you should include them anywhere that they will enhance clarity.

The rightmost argument is on top of the stack, with any following arguments beneath it. In other words, arguments are pushed onto the stack in left to right order, leaving the most recent one (the rightmost one in the diagram) on the top.

Following the stack comment in the preceding example are a series of words that describe the behavior of `dac!`. Executing `dac!` is the same as executing the list of words in its colon definition.

Note that FCode words are separated by spaces, tabs, or newlines; "`( data ″` is *not* the same as "`(data ″`. Any visible character is part of a word, and not a separator.

While case is not significant, by convention FCode is written in lower case.

## Programming Style

Some people have described Forth as a write-only language. While it sometimes ends up that way, it *is* possible to write Forth (and FCode) programs that can be read and understood by more than just the original programmer.

### Commenting Code

*Comment code extravagantly*, then consider adding more comments. The comments can help you and others maintain your code, and they don't add to the final size of the resulting FCode PROM.

### Short Definitions

*Keep word definitions short.* If your definition exceeds half a page, see if you can break it up into two or more definitions. If it grows to a page or longer, you *should* break it up, if only to make the code easier to support in the future.

A *good* size for a word definition is one or two lines of code.

### Stack Comments

*Always include stack comments in word definitions.* It can be useful to compare intended function with what the code really does. Here's an example of a word definition with acceptable style

```
\ xyz-map  establishes a virtual-to-physical mapping for each of the
\ addressable regions on the board (except for the FCode PROM, which is
\ handled automatically by the Boot PROM).  my-address is a predefined
\ procedure that returns the physical address of the slot in which
\ this board is located (as determined by the Boot PROM).

: xyz-map  ( -- )

\ Base-address  Offset  Size  create-mapping remember-virtual-addr

   my-address  40.0000 + 4 map-sbus  ( virtaddr )
   is status-register ( )
   my-address  80.0000 + frame-buf-size  map-sbus ( virtaddr )
   is frame-buffer-adr
;
```

## A Minimum FCode Program

If an SBus card is not needed during the boot process, a minimal FCode program that merely declares the name of the device will often suffice. Here is an example of an acceptable minimum program:

```
FCode-version1
" SUNW,bison"  xdrstring  " name" attribute
end0
```

This program, using just 31 bytes of PROM space, creates a "`name`" property called "`SUNW,bison`" that will be used by the SunOS driver's `identify` routine to identify this device. The name that you use should always begin with your company name.

**Note**: To avoid name conflicts between different companies' products, use your company's public stock symbol.

You can also use the following shorthand form. The FCode program generated will be identical to the minimum program given above.

```
FCode-version1
" SUNW,bison" name
end0
```

You might also want to include additional code to declare additional properties, create selftest routines, or to initialize the device after power-on. For more information, see Chapter 3 (Producing FCode).

## FCode Classes

There are four general classes of FCode source words:

- **Primitives.** These words generally correspond directly to conventional Forth words, and implement functions such as addition, stack manipulation, and control structures.
- **System.** These are extension words implemented in the boot PROMs, and implement functions such as memory allocation and device attribute reporting.
- **Interface.** These are specific to particular types of SBus devices, and implement functions such as `draw-character`.
- **Local.** These are private words definitions, implemented and used by SBus devices.

Each FCode primitive is represented in the SBus card's PROM as a single byte. Other FCodes are represented in the SBus PROM as two consecutive bytes. The first byte, a value from 1 to 15, may be thought of as an escape code.

One-byte FCode numbers range in value from `0x10` to `0xfe`. Two-byte FCode numbers begin with a byte in the range `0x01` to `0x0f`, and end with a byte in the range `0x00` to `0xff`. The single-byte values `0x00` and `0xff` signify "end of program" (either value will do; conventionally, 0x00 is used):

Currently-defined FCodes are listed according to both functional groups and in numeric order in Appendix A, *FCode Reference*.

### Primitive FCodes

There are more than 300 primitive FCode words, most of which exactly parallel Forth-83 words, divided into three groups:

- FCode words that generate a single FCode byte
- cross-compiler equivalents
- cross-compiler directives

Primitive FCode words that have an exact parallel with standard Forth-83 words are given the same name as the equivalent Forth-83 word. Appendix B contains further descriptions of primitive FCodes.

There are about another 70 **cross-compiler equivalents**, most of which also have direct Forth-83 equivalents. These are convenient source code words translated by the tokenizer into short sequences of FCode primitives.

**cross-compiler directives** are words that generate no FCodes, but are used to control the interpretation process. Cross-compiler directives include the words

- `binary`, `decimal`, `hex`, and `octal`
- `b#`, `d#`, `h#`, and `o#`
- `headers and headerless`
- `\ and (`
- `.(`
- `alias`

## System FCodes

System FCodes are used by all classes of SBus FCode drivers for various system-related functions. System FCodes may be either **service** words or **configuration** words.

- Service words are available to the SBus device's FCode driver when needed for functions such as memory mapping or diagnostic routines.
- Configuration words are included in the driver to document characteristics of the driver itself. These "properties" are passed up to the device's SunOS driver.

## Interface FCodes

Interface FCodes are standard routines that use the workstation's CPU to perform the device. Different classes of devices will each use only the appropriate set of interface FCodes.

For example, if the system wants to paint a character on the display screen, it does it by calling the interface FCode routine DRAW-CHARACTER. This requires the frame buffer's FCode driver to assign its own definition into the DRAW-CHARACTER interface word. It does this as follows:

```
: MY-DRAW ( char -- ) \ "Local" word to draw a character.
    ...               \ Definition contents.
;                     \ End of MY-DRAW definition.
: MY-INSTALL ( -- )   \ Local word to install all interfaces.
    ...
    ['] MY-DRAW IS DRAW-CHARACTER
    ...
;
```

When MY-INSTALL executes, DRAW-CHARACTER has the behavior of MY-DRAW.

## Local FCodes

Local FCodes are assigned, where needed, to words defined within the body of SBus driver code. There are over 2000 FCode byte values allocated for local FCodes. The byte values are meaningful only within the context of a particular driver. Different drivers reuse the same set of byte values.

# 3

# *Producing FCode*

## FCode Production Tools

A number of software tools are required or recommended for you to use to develop FCode programs for SBus cards. They include the following:

- A text editor. The editor must be able to save text in the form of plain ASCII text files on any SPARCstation or Sun-4. Pick your favorite.
- The tokenizer program. This converts Forth programs in the form of text files into FCode files. The tokenizer runs on SPARCstations or Sun-4's and is available from the Sun SBus Support Group.
- Tools for downloading program files to the target machine (the machine that the SBus card will be plugged into for testing). Download programs include `dl`, `dlbin`, `dload` and `boot`, and are available in the Open Boot PROM itself.
- Other tools, such as `pburn`, a PROM burner program, or `reheader.fth` are available from the Sun SBus Support Group.

## Developing FCode Drivers

Part of the work of developing SBus devices includes writing, testing, and installing FCode drivers for the device. These drivers serve three functions:

- ❑ To identify the device and its characteristics, required for all SBus devices
- ❑ Optionally, to exercise the device during development, to verify its functionality
- ❑ To provide the necessary driver to be used by the boot PROM during power-up, appropriate only for those SBus devices actually used during booting

**The main steps in developing FCode drivers for an SBus device are as follows:**

1. Configure the target machine.
2. Write the ASCII Forth text. (Optionally converting the ASCII Forth into FCode with the Tokenizer.)
3. Give commands as necessary to prepare for the Forth download.
4. Place the ASCII Forth or FCode into a known memory location on the target machine.
5. Interpret the Forth code at the known memory location, thus creating definitions which may be executed as desired.
6. Execute Forth definitions as desired, to test the device and the Forth code.
7. If a Forth error is discovered, or additional code is needed, go back to step 2, edit the ASCII Forth text file, and repeat.

The rest of this chapter describes how to write, test, and install SBus device FCode drivers.

## Configuring the Target Machine

**Configuring the target machine requires four main steps:**

- ❏ Set the NVRAM parameter `fcode-debug?` to `true`.
- ❏ Set the NVRAM parameter `sbus-probe-list` to skip (don't probe) the slot which will contain the SBus device being tested.
- ❏ Power-down the target system, and plug the SBus card into the chosen slot.
- ❏ Power-up the target system, and enter the Open Boot PROM Toolkit (ok prompt).

### Setting NVRAM Parameters

Two NVRAM parameters, `fcode-debug?` and `sbus-probe-list`, need to be set:

Set them from the PROM monitor, as follows:

```
ok setenv fcode-debug? true
ok
ok setenv sbus-probe-list 013
ok
```

Setting `fcode-debug?` to `true` saves the names of all words created by interpreting FCode.When `fcode-debug?` is set to `false` (the default state), name field headers are discarded to save RAM space. If the name field headers are saved, you can execute any of the newly-defined words by typing its name at the `ok` prompt.

The value of `sbus-probe-list` defines which SBus card slots will be probed during start up. Until your SBus card has a working FCode PROM on it, you won't want it to be probed.

For example, setting sbus-probe-list to 013 directs the boot PROM during startup to first probe (interpret FCode) SBus slots 0 (built-in devices), then slot 1, and finally slot 3. This leaves SBus slot 2 unprobed, free for use by your device.

On the SPARCstation1 and SPARCstation1+, slots 1 and 2 are general-purpose. Slot 3 may be used *only* for DMA slave

devices, such as framebuffers. It may *not* be used for DMA master devices, such as disk drives or network interfaces.

Once you have a working FCode PROM on the SBus device, and want to do a full system test (including automatic probing of the new device), you'll want to restore `sbus-probe-list` to its normal value of 0123.

If you're working on a SPARCstation model with more or less than three SBus slots, alter the value of `sbus-probe-list` as needed.

## Plugging in the SBus Card

After setting the NVRAM parameters, power-down the target system, and plug the SBus device into the appropriate slot.

After you've plugged the SBus card in to the proper slot, and after you've checked all appropriate connections, power-up the target system, and enter the Toolkit (`ok` prompt).

```
Type b (boot), c (continue), or n (new command mode)
>n
ok
```

If the system tries to auto-boot, interrupt it by pressing L1-A. If the Sunmon-compatible prompt appears, type `n` Return at the prompt:

You can disable auto-boot with:

```
ok setenv auto-boot? false
ok
```

You can disable the Sunmon-compatible prompt with:

```
ok setenv sunmon-compat? false
ok
```

If you are developing a framebuffer, you will want to use `ttya` as your console. Set this up with:

```
ok ttya io
```

The responding "ok" will be echoed over serial line `ttya`.

If you want all post-power-on dialogues to pass over ttya after the next power-on reset, you can issue:

```
ok  setenv input-device ttya
ok  setenv output-device ttya
ok
```

## Writing the Forth Program

Use your favorite text editor, under SunOS, to create a text file containing the desired test ASCII Forth code. The file should begin with:

```
\ Title comment.  This describes the file that follows.
FCode-version1
hex
```

and should end with the line:

```
end0
```

For a non-boot device, some simple declarations of the device name and other attributes are all that is needed. For example:

```
\ "Bull" Printer FCode
FCode-version1
hex
" SUNW,bullprinter" name      \ Declare the "name" attribute
                              \ If this line isn't present, SunOS
                              \ may not boot properly.
" SUNW,501-1419-01" model     \ Declare the "model" attribute
3  0  intr                    \ Generates the "intr" (interrupt)
                              \ attribute of SBus#3 interrupt
```

```
my-address 8000 +   my-space   20  reg
                               \ Declares "reg" attribute, showing
                               \ that device registers are at 8000-801f
                               \ offset from the start of the card
end0
```

Other statements that might have been included are the following:

```
\ Create a string, then discard it.  An easy way to put text information
\ into an FCode PROM.
" Copyright 1990 Sun Microsystems, Inc."  2drop   \ Copyright

" @(#)bullprint.fth  1.1  90/03/08"  2drop  \ Source control identifier

\ Additional attribute for dots/inch
d# 100 xdrint   " dpi"    attribute

\ Additional attribute for hypothetical "test string"
" abcdefg" xdrstring    " teststring"   attribute
```

Typical code for a boot device consists of a series of colon definitions, building on each other and ending with something like:

```
...
: my-probe-routine  ( -- )
...
['] my-install-routine  is-install
['] my-remove-routine   is-remove
['] my-selftest-routine is-selftest
...
;
my-probe-routine   ( executes the above definition )
end0
```

If you need to declare multiple register fields, do *not* call the `REG` command multiple times. Instead, build a multi-value "`reg`" attribute, as follows:

```
my-address  4000  +  my-space  xdrphys       \ Create offset#1.
20  xdrint  xdr+                             \ create size#1 and merge.
my-address 20.0000 + my-space xdrphys xdrt+  \ create offset#2 and merge
1000 xdrint xdr+                             \ create size#2 and merge.
" reg" attribute
```

If you need to declare multiple interrupts, do *not* call the INTR command multiple times. Instead, build up a multi-value "`intr`" attribute, as follows:

```
3 sbus-intr>cpu xdrint              \ create SBus interrupt#1.
0 xdrint xdrt+                      \ create null vector#1 and merge.
5 sbus-intr>cpu xdrint xdrt+        \ create SBus interrupt#2 and merge.
0 xdrint xdrt+                      \ create null vector#2 and merge.
" intr" attribute
```

A boot device must contain a **device-type attribute**, such as:

```
" display" device-type
\ Recognized values are:  display, disk, tape or network
```

## Converting Forth to FCode

The tokenizer program, available from the Sun SBus Support Group, translates the test ASCII Forth program into FCode. Instructions for running the tokenizer are provided along with the tokenizer.

If you want to, you can skip using the tokenizer and download and execute ASCII Forth text directly. However, this may cause problems since some commands do not have name headers in the boot PROM, and will thus be unrecognized.

A file named `reheader.fth`, available from the Sun SBus Support Group, when downloaded and executed provides the missing words if you prefer not to use the tokenizer.

## Preparing for the Download

Now that you have your Forth (or FCode) program ready for testing, you need to download it to the target machine so you can begin testing it.

Set the value of `my-address` before downloading Forth code. To declare the hardware location of the SBus device being tested, enter the following:

```
ok 200.0000 is my-address
ok
```

Appropriate hexadecimal values for SPARCstation1 and 1+ SBus slots are:

- 200.0000 Slot 1
- 400.0000 Slot 2
- 600.0000 Slot 3

The decimal point shown in the hexadecimal address values is optional, and is for readability only.

Prepare the device tree for a new entry by issuing the `new-slot-node` command:

```
ok new-slot-node
ok
```

`new-slot-node` is not in boot PROM versions 1.0 or 1.1, but you can download it with the `reheader.fth` file.

When an FCode PROM is probed during power-up or when FCode interpretation is done manually from an FCode PROM by `probe-slot`, both `my-address` and `new-slot-node` are automatically performed. In either of these cases, initialization commands are unnecessary.

## Downloading to the Target Machine

With your Forth program written, you can place the ASCII Forth or F memory location on the target mac

There are several ways to do this:

- Use the FCode contained in the SBus device's PROM. (The job has already been done for you.)
- Download the file over Ethernet.
- Download the file over a serial line.
- Download the file from the hard disk.
- Download the file from a floppy disk.

## Using an FCode PROM

Typically, this FCode PROM method is only used *after* the code has been downloaded and debugged using one of the other methods described in this manual.

You can use the `pburn` utility (available from the SBus Support Group) together with a Data I/O PROM Burner to create your SBus device PROM. (You can use other systems as long as they are capable of creating a PROM from a binary file.)

If you want to interpret the PROM using `byte-load`, you will need to first ***map*** the SBus device, to create a usable virtual address for accessing the SBus PROM data, as follows:

```
ok 200.0000 1000 map-sbus
ok
```

In the above example,

- 200.0000 is the value for SBus slot 1.
- 1000 is the amount of memory to be mapped, in this case, 1 page. Use a larger value if the PROM data occupies more space than this.
- The virtual address generated by `map-sbus` is left on the stack.

To check that the mapping and memory access worked, enter the following:

```
ok dup 20 dump
ffd1e000  fd 00 92 e6 00 00 ...
ffd1e010  67 74 68 72 65 01 ...
ok
```

The first byte of data returned should always be `fd`, indicating a valid FCode.

**Downloading from Ethernet**

To download the file over Ethernet, either use the `dload` or `boot -h` commands as described below.

```
ok 4000 dload /path/filename.ext
ok
```

The file name should specify the full path relative to the server's root. `dload` places the contents of the file at memory location 4000 (hex), a recommended location to which to download files.

**Note**: `dload` uses the `tftp` protocol. The appropriate server machine may need to have its permissions adjusted before this will work. See the *Open Boot PROM Toolkit User's Guide* (Downloading Files) for more information.

An alternative to `dload`, is the `boot -h` command::

```
ok boot net filename.ext -h
ok
```

In this case, the file name is relative to the client's root on its server. Thus, specifying *filename.ext* will access a file such as `/export/root/<client>/filename.ext` .

To download a Forth or FCode file, first use `fakeboot` to convert it into an `a.out`-format file of the proper length before

using `boot -h`. (`fakeboot` is available from the SBus Support Group.)

**Downloading from a Serial Line**

To download the file over a serial line, you need a machine at the other end of the serial line that can transfer a file on request. One method is to set up a `tip` window on a different Sun system, and to use it like a dumb terminal.

Examples assume a `tip` window setup, although you could use other means. See the *Open Boot PROM Toolkit User's Guide* (Downloading Files) for more information on setting up a `tip` window.

The normal downloading rate over a serial line is 9600 baud. If you need to use a different speed, use `ttya-mode` to change the serial port A speed, and `ttyb-mode` to change the speed of serial port B.

Use the `dl` command to download an ASCII Forth text file:

```
ok dl
```

Now send the file, pressing Control-D to signal the end of the file transfer. An example of doing this with `tip` is::

```
~C (Local command?) cat filename.fth
(Away for 2 seconds)
(!)
^D
ok
```

The file is automatically interpreted after the download ends.

If you see any unexpected output after sending the `^D` when you use `dl`, check your `.cshrc` file: `cat` opens a sub-shell, and thus invokes your `.cshrc` file. If your `.cshrc` file generate any output text, this text will be prepended to your download file. This generates strange results and usually corrupts your file.

For an FCode file (in the form of an `a.out` binary file), use the `dlbin` command:

```
ok dlbin
```

Now, send the file. For example:

```
~C (Local command?) sendbin filename.o
(Away for 2 seconds)
(!)
ok
```

The FCode is then left in memory, starting at an address determined by the FCode `a.out` file header, usually 0 or 4000, hexadecimal. Use `dump` to verify a proper download and its location.

Early versions of the tokenizer program created an a.out header leaving the file at address 0. Versions 1.1 and later of the tokenizer leave the FCode file at address 4000.

The `sendbin` command is a shell script which you should have previously defined on the Sun system being used as the `tip` window. This script is defined as follows for SunOS version 4.0.3 or later:

```
#!/bin/sh
sleep 5 >/dev/tty &
stty -parenb cs8 raw -echo
cat $1
```

### Downloading from a Disk

To download the file from the hard command, as follows:

```
ok boot disk filename.ext -h
ok
```

While the file should reside in the root directory of the disk it's stored on, you must omit the leading "/" in the file name when you invoke `boot` .

To download a Forth or FCode file with `boot -h` , first convert the file to the `a.out` form with `fakeboot`.

**Downloading from a Diskette**

To download the file from a diskette, use the `boot -h` command, as follows:

```
ok boot fd() filename.ext -h
ok
```

The file should reside in the diskette's root directory, but you should omit the leading "/" of its pathname when you invoke `boot` .

Build a bootable diskette this way:

1. Use a 1.4Mb HD diskette, *not* a DD diskette
2. Format the diskette with `/bin/fdformat`
3. Create a filesystem on the diskette with `/usr/etc/newfs /dev/rfd0a`
4. Mount the diskette with `mount /dev/fd0a /mnt`
5. Copy any `/boot` program to `/mnt` from either your local disk or any server.
6. Install a boot block using

```
% cd /usr/mdec
% installboot /mnt/boot bootfd /dev/rfd0a
```

To download a Forth or FCode with `boot -h` , first convert the file to the `a.out` form with `fakeboot`.

Now copy the file to the diskette with:

```
%  cp filename.ext  /mnt
```

A file now copies into `/mnt/filename.ext` will appear on the diskette in its root directory as `/filename.ext` .

## Running Downloaded Code

Now you can begin to interpret the Forth code at the known memory location, thus creating definitions which may be executed as desired.

There are several different methods of interpreting the Forth code, depending on the type of file (ASCII Forth text, or binary FCode), and depending on the method of download used:

- `(addr) 1 byte-load` interprets binary FCode at the given address.
- `4000 file-size @ eval` interprets ASCII Forth text from a file downloaded into address 4000 using `dload`.
- `4010 4004 @ eval` interprets ASCII Forth text from a file downloaded using `boot -h` (and converted using `fakeboot`).
- ASCII Forth text downloaded using `dl` is interpreted automatically and no further command is needed.
- If an SBus device PROM is present, and probing is enabled for the given slot, then the FCode on the device PROM is interpreted during power-on initialization. If probing for that slot was disabled, manually probe the slot with `<slot#> probe-slot`

The following sections describe each of these methods in more detail.

(`addr) 1 byte-load` interprets b **Using** `byte-load`
address. The value for `(addr)`
following:

- `0` or `4000` for FCode downloaded using `dlbin` (4000 for FCode from `tokenizer` versions 1.1 or later)
- `4000` for FCode downloaded into address 4000 using `dload`

- `4030` for FCode converted with `fakeboot` and downloaded with `boot -h`
- `(virtaddr)`, the virtual address returned by `map-sbus`, when FCode is already present on an SBus device PROM

The 1 in the phrase `(addr) 1 byte-load` indicates the spacing of the FCode bytes. This will usually be 1; read FCode bytes consecutively from locations `(addr)`, `(addr+1)`, `(addr+2)`, `(addr+3)`, and so on.

For some unusual devices, this could be different, for example, 4 would direct `byte-load` to read FCode bytes consecutively from locations `(addr)`, `(addr+4)`, `(addr+8)`, `(addr+12)`, `...` A value of 0 means to obtain consecutive FCode bytes by reading location `(addr)` over and over again.

**Using** `eval`

`eval` interprets ASCII Forth text at the given address. Its usage depends on the downloading method you chose to use.

`4000 file-size @ eval` interprets ASCII Forth text from a file downloaded into address 4000 by `dload`.

`file-size @` returns the size of the file just downloaded. (`eval` expects an `addr len` on the stack to specify where and how much ASCII Forth text to interpret.)

`4010 4004 @ eval` interprets ASCII Forth text from a file downloaded using `boot -h` (and converted by `fakeboot`).

`fakeboot` processes the text file into an `a.out` format. The text begins at location 4010 after being downloaded with `boot -h`, and the file size is stored in location 4004.

**Manually Probing SBus Slots**

Probing an SBus slot interprets the FCode PROM of the device plugge
has been disabled, you can manua
command "`<slot#> probe-slot`". For example, "`2 slot-`

`probe`" probes the PROM in slot 2. `probe-slot`'s operation is basically equivalent to this:

```
ok 400.0000 is my-address
ok new-slot-node
ok my-address 1.0000 map-sbus (virt)
ok constant slotloc\ save address
ok slotloc 1 byte-load
ok slotloc 1.0000 free-virtual\ optional
```

## Testing Downloaded Code

You can now execute Forth definitions as desired, to test the device and the Forth code.

There are a wide variety of procedures that could be executed at this point, depending on your needs.

- ❏ The command `words` shows the names of all words in the dictionary. You should see the word names you just created.
- ❏ The command `see` *wordname* decompiles the word *wordname*. Use this to verify that different words that you've created are defined correctly.
- ❏ Any defined words may be invoked individually.
- ❏ To test a framebuffer, execute:

```
ok my-install   (your install routine which was given to "is-install")
ok reset-emulator   (reset terminal emulator)
ok " testing" ansi-type   (optional, send a string to framebuffer)
ok screen output
(ok prompt should now appear on target framebuffer)
```

If your tests uncover any programming errors, or you want to add new functionality, you can do any of the following:

- ❏ Isolate the error. Execute your code step-by-step by typing the words. This is much easier if your code has been

written as a number of short definitions. Enable `showstack` after each step to check the stack for correctness.

- ❑ Edit the original ASCII Forth text file. Translate, download, and test the new version.
- ❑ Type new word definitions from the monitor and test them.
- ❑ Use `patch` to edit word definitions in memory and test them.

# 4

# *FCode Dictionary*

## Introduction

This dictionary describes the pre-defined FCode words that you can use as part of FCode source code programs. Appendix A, "FCode Reference", contains a command summary, with words grouped by function.

The words are given alphabetically in this chapter, sorted by the first alpahbetic character in the word's name. So, for example, the words `mod` and `*/mod` are adjacent to each other. Words having no alphabetic characters in their names are placed at the beginning of the chapter, in ASCII order.

FCodes names are given in upper case for clarity. The boot PROM and tokenizer are case-insensitive (all Forth words are converted to lower-case internally). The only exceptions are literal text, such as text inside of " strings and text arguments to the ASCII command, which are left in the original form. In general, you may use either upper- or lower-case.

Words described as **head missing** do not have the name fields present in some boot PROMs. Thus, although the function can be properly tokenized and executed, it cannot be typed interactively from the Toolkit "`ok`" prompt, and it cannot be downloaded and executed in source (text) form. `reheader.fth` will restore the name fields for all such "head missing" words,

so that they may be typed interactively or downloaded in source form.

Branch offsets (such as generated by `IF` statements, `BEGIN` loops or `CASE` statements) are encoded as a single, signed byte, its value falling within the range of +/-127.

If you create large (>50 word) definitions, it is possible to exceed this branch limit and generate erroneous results. If necessary, `OFFSET16` encodes branch offsets with two bytes instead of one, thus eliminating this problem. However, it is bettter to simply create shorter definitions instead. See `OFFSET16` for more details.

Most numbers are encoded using `B(LIT)` . See `B(LIT)` for more details.

All arithmetic uses 32-bit signed values, unless otherwise specified.

Defining words create a header by calling either `NAMED-TOKEN` or `NEW-TOKEN` . See `NAMED-TOKEN` and `NEW-TOKEN` for more details.

All FCode byte values listed in this chapter are given in hexadecimal.

All words applying *only* to display `DEVICE-TYPE` cards (such as framebuffers) are not listed here but are described in Appendix B, “Framebuffer FCodes”. Other sections may be added in the future for other specific `DEVICE-TYPE`s (such as network or disk types).

## FCode Definitions

The rest of this chapter contains definitions of the FCodes defined for use in the SPARCstation Open Boot PROM.

**!**

```
!           ( n adr -- )
code# 72
```

Store the 32-bit value `n` at `adr`. For more portable code, use `L@` if you explicitly want a 32-bit access.

**"**

```
"  (text)"     ( -- adr len )
generates:  B(") len text
code# 12 len xx xx xx ...
```

This word is used to compile a text string, delimited by an " . At execution time, the address and length of the string is left on the stack.

**#**

```
#  ( +L1 -- +L2 )
code# 99
```

The remainder of `+l1` divided by the value of `BASE` is converted to an ASCII character and appended to the output string toward lower memory addresses. `+l2` is the quotient and is maintained for further processing. Typically used between `<#` and `#>` . See `(.)` and `(U.)` for typical usages.

**#>**

```
#>  ( L -- adr +n )
code# 97
```

Pictured numeric output conversion is ended dropping `L`. `adr` is the address of the resulting output array. `+n` is the number of characters in the output array. `adr` and `+n` together are suitable for `TYPE` . See `(.)` and `(U.)` for typical usages.

**'**

```
'  name   ( -- acf )
generates:  B(')
code# 11 FCode(name)
```

Used to generate the code field address (acf) of the word immediately following the `' . '` should only be used *outside* of definitions. See `[']` for more details.

**(**

```
( text)  ( -- )
code# none
```

Ignore subsequent text after the "`(`" up to a delimiting ")" . Note that a space is required after the `(` . Although either `(` or `\` may be used equally well for documentation, by common convention we use `( ... )` for stack comments and `\ ...` for all other text comments and documentation. See also `(S` .

**(.)**

```
(.)          ( n -- adr len )
generates:  DUP ABS <# #S SWAP SIGN #>
code#       47  2d  96 9a 49   98   97
```

This is the numeric conversion primitive, used to implement display words such as "." It converts a number into a string. If n is negative, the first character in the array will be a minus (-) sign.

*****

```
*  ( n1 n2 -- n3 )
code# 20
```

`n3` is the arithmetic product of `n1` times `n2`. If the result cannot be represented in one stack entry, the least significant bits are kept.

***/**

```
*/  ( n1 n2 n3 -- n4 )
generates:  >R * R> /
code# 30 20 31 21
```

Calculates `n1*n2/n3`. The inputs, outputs and intermediate products are all 32-bit.

**+**

```
+   ( n1 n2 -- n3 )
code# 1e
```

`n3` is the arithmetic sum of `n1` plus `n2`.

**+!**

```
+!  ( n adr -- )
code# 6c
```

`n` is added to the 32-bit value stored at `adr`. This sum replaces the original value at `adr`.

**,**

```
,     ( n -- )
code# d3
```

Compile a number into the dictionary. In this system, the number of bytes compiled is 4 (same as `L,`). See `C,` for limitations.

**-**

```
-    ( n1 n2 -- n3 )
code# 1f
```

`n3` is the result of subtracting `n1` minus `n2`.

**-1**

```
-1   ( -- -1 )
code# a4
```

Leave the value -1 on the stack. The only numbers which are not encoded using `B(LIT)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to save space.

**.**

```
.   ( n -- )
code# 9d
```

The absolute value of `n` is displayed in a free field format with a leading minus sign if `n` is negative, and a trailing space.

**Note**: If the base is hexadecimal, . displays the number in unsigned format, since signed hex display is hardly ever what you want. Use `S.` when you want to display signed hex numbers. See also `S.` .

**." text"**

```
."  text"    ( -- )
generates:  B(") len text          TYPE
code#       12   len xx xx xx ... 90
```

This word compiles a text string, delimited by `"`. At execution time, the string is displayed, for example, in `." hello world"`

This word is equivalent to using `" text"  TYPE`

`."` is normally used only within a definition. The text string will be displayed later when that definition is called. You may wish to follow it with CR to flush out the text buffer immediately.

Use `.(` to print anything while the FCode PROM is being interpreted.

See `TOKENIZER[` for details about printing at tokenize time.

**.( text)**

```
.( text)         ( -- )
code# none
generates:  B(")  len text          TYPE
code#       12    len xx xx xx ... 90
```

Gathers a text string, delimited by `)`, to be immediately displayed. For example: `.( hello world)`

This word is equivalent to: `" text"  TYPE`

Use this to print out text at the time the FCode PROM is being interpreted (you may wish to follow it with a `CR` to flush out the text buffer immediately). This word should only be called `outside` of definitions.

Note that the string will typically be printed out of serial port A, since any framebuffer present may not yet be activated at the time that SBus slots are being probed. Use `."` for any printing to be done when new words are later executed.

See `TOKENIZER[` for details about printing at tokenize time.

**/**

```
/   ( n1 n2 -- quot )
code# 21
```

Calculates `n1` divided by `n2`. An error condition results if the divisor (`n2`) is zero.

**0**

```
0  ( -- 0 )
code# a5
```

Leave the value 0 on the stack. The only numbers which are not encoded using `B(LIT)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to save space.

**0<**

```
0<  ( n -- flag )
code# 36
```

Flag is true if `n` is less than zero (negative).

**0<=**

```
0<=  ( n -- flag )
code# 37
```

Flag is true if n is less than or equal to zero.

**0=**

```
0=  ( n -- flag )
code# 34
```

Flag is true if `n` is zero. This word will invert any flag.

**0<>**

```
0<>  ( n -- flag )
code# 35
```

Flag is true if `n` is not zero.

**0>**

```
0>  ( n -- flag )
code# 38
```

Flag is true if n is greater than zero.

**0>=**

```
0>=  ( n -- flag )
code# 39
```

Flag is true if `n` is greater than or equal to zero.

**1**

```
1  ( -- 1 )
code# a6
```

Leave the value 1 on the stack. The only numbers which are not encoded using `B(LIT)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to save space.

**1+**

```
1+  ( n1 -- n2 )
generates:  1 +
code# a6 1e
```

`n2` is the result of adding one to `n1`.

**1-**

```
1-  ( n1 -- n2 )
generates:  1 -
code# a6 1f
```

`n2` is the result of subtracting one from `n1`.

**2**

```
2   ( -- 2 )
code# a7
```

Leaves the value 2 on the stack. The only numbers which are not encoded using `B(LIT)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to save space.

**2!**

```
2!  ( n1 n2 adr -- )
code# 77
```

`n1` and `n2` are stored in consecutive 32-bit locations starting at `adr`. `n2` is stored at the lower address.

**2***

```
2*  ( n1 -- n2 )
code# 59
```

`n2` is the result of shifting `n1` left one bit. A zero is shifted into the vacated bit position. This is equivalent to multiplying by 2.

**2+**

```
2+  ( n1 -- n2 )
generates:  2 +
code# a7 1e
```

`n2` is the result of adding 2 to `n1`.

**2-**

```
2-  ( n1 -- n2 )generates:  2 -
code# a7 1f
```

`n2` is the result of subtracting 2 from `n1`.

**2/**

```
2/  ( n1 -- n2 )
code# 57
```

`n2` is the result of arithmetically shifting `n1` right one bit. The sign is included in the shift and remains unchanged. This is equivalent to dividing by 2.

**2@**

```
2@  ( adr -- n1 n2 )
code# 76
```

`n1` and `n2` are two numbers stored in consecutive 32-bit locations starting at `adr`. `n2` is the number which was stored at the lower address.

**3**

```
3   ( -- 3 )
code# a8
```

Leaves the value 3 on the stack. The only numbers which are not encoded using `B(LIT)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to save space.

**:**

```
:  name   ( -- )         at creation
          ( ?? -- ?? )   at execution
generates:  new header, B(type) = B(:)
code# (header) b7
```

Begin a new definition, terminated by `;` Used in the form:

```
:  newname  ...  ;
```

Later usage of *newname* is equivalent to usage of the contents of the definition. See `NAMED-TOKEN`, `NEW-TOKEN` for more information on header formats.

**;**

```
;  ( -- )
generates:  B(;)
code# c2
```

Ends the compilation of a colon definition. See also `:`.

**<**

```
<  ( n1 n2 -- flag )
code# 3a
```

Flag is true if `n1` is less than `n2`. `n1` and `n2` are signed integers.

**<#**

```
<#  ( -- )
code# 96
```

Initialize pictured numeric output conversion. The words: `<# # #S HOLD SIGN #>` can be used to specify the conversion of a 32-bit number into an ASCII character string stored in right-to-left order. See `(.)` and `(U.)` for typical usages.

**<<**

```
<<  ( n1 +n -- n2 )
code# 27
```

`n2` is the result of logically left shifting `n1` by `+n` places. Zeroes are shifted into the least-significant bits.

**<=**

```
<=  ( n1 n2 -- flag )
code# 43
```

Flag is true if `n1` is less than or equal to `n2`. `n1` and `n2` are signed integers.

**<>**

```
<>  ( n1 n2 -- flag )
code# 3d
```

Flag is true if `n1` is not equal to `n2`. `n1` and `n2` are signed integers.

**=**

```
=  ( n1 n2 -- flag )
code# 3c
```

Flag is true if `n1` is equal to `n2`. `n1` and `n2` are signed integers.

**>**

```
>  ( n1 n2 -- flag )
code# 3b
```

Flag is true if `n1` is greater than `n2`. `n1` and `n2` are signed integers.

**>=**

```
>=  ( n1 n2 -- flag )
code# 42
```

Flag is true if `n1` is greater than or equal to `n2`. `n1` and `n2` are signed integers.

**>>**

```
>>  ( n1 +n -- n2 )
code# 28
```

`n2` is the result of logically right shifting `n1` by `+n` places. Zeroes are shifted into the most-significant bits. Use `>>A` for signed shifting.

**?**

```
?  ( adr -- )
generates:  @ .
code# 6d 9d
```

Fetch and print the 32-bit value at the given address. An old standard Forth word, primarily used interactively.

**@**

```
@  ( adr -- n )
code# 6d
```

`n` is the 32-bit value stored at `adr`. For more portable code, use `L@` if you explicitly want a 32-bit access.

**[']**

```
[']  name   ( -- acf )
generates:  B(')
code# 11 FCode(name)
```

`'` or `[']` are used to generate the code field address (acf) of the word immediately following the `'` or `[']`.

`'` should be used *outside* of definitions; `[']` should be used *inside* definitions. For example:

```
: my-probe...   ['] my-install  is-install...;  or
' my-install  is-install
```

Examples shown may use either ' or ['], depending on the context. They both behave identically in nearly all circumstances, but it is vital to use the correct one in the correct place, i.e. ['] within definitions and ' outside of definitions.

**Note**: In normal Forth, ' may be used within definitions for the creation of language extensions, but such usage is not applicable to FCode programs.

## \

```
\ rest-of-line  ( -- )
code# none
```

Ignore the rest of the input line after the \. It can occur anywhere on an input line. Note that a space must be present after the \. See ( or (S for another form for delimiting comments.

## <<A

```
<<A  ( n1 +n -- n2 )
generates:  <<
code# 27
```

Arithmetic left-shift (left-shift with sign-extend), to round out the existing words <<, >>, and >>a . This word is useless, because the carry out from an arithmetic left shift is not accessible later.

## >>A

```
>>A  ( n1 +n -- n2 )
code# 29
```

n2 is the result of arithmetically right shifting n1 by +n places. The sign bit of n1 is shifted into the most-significant bits.

## ABS

```
ABS  ( n -- u )
code# 2d
```

`u` is the absolute value of `n`. If `n` is the maximum negative number, `u` is the same value (since the maximum negative number in two's complement notation has no positive equivalent).

## AGAIN

```
AGAIN  ( -- )
generates:  BBRANCH -offset
code# 13 offset
```

Used in the form: `BEGIN ... AGAIN` to generate an infinite loop. Press L1-A to exit from this loop. *Use this word with caution!*

## ALIAS

```
ALIAS new-name old-name   ( -- )
code# none
```

`ALIAS` creates a new name, with the exact behavior of some other existing name. The new name may then be used interchangeably with the old name, having the same effect.

The tokenizer does *not* generate any FCode for an `ALIAS` command, but instead simply updates its own lookup table of existing words. Any occurance of the new word causes the assigned FCode value of the *old* word to be generated. One implication is that the *new* word will not appear in the Toolkit dictionary after the FCode program is compiled.

If the original FCode source text is downloaded and interpreted directly, without being tokenized or detokenized, then any new `ALIAS` words *will* show up and be usable directly.

## ALIGNED

```
ALIGNED   ( adr1 -- adr2 )
code# ae
```

Increase `adr1` to the next machine word boundary — to the next value evenly divisible by 4. The correct boundary could vary on other CPU implementations.

## ALLOC-MEM

```
ALLOC-MEM   ( #bytes -- virtual )
code# 8b
```

Allocate some free physical memory from Forth, and return its virtual address. This should only be used to allocate small amounts of memory — a few hundred bytes or less. `DMA-ALLOC` is better for large amounts.

## AND

```
AND  ( n1 n2 -- n3 )
code# 23
```

`n3` is the bit-by-bit logical `AND` of `n1` with `n2`.

## ASCII

```
ASCII x  ( -- n )
generates:  B(LIT) value
code# 10 00 00 00 xx
```

Interpret the next letter as an ASCII code. For example:

```
ASCII C (equals hex 43)
ASCII c (equals hex 63)
```

## ATTRIBUTE

```
ATTRIBUTE ( value-xdr-adr value-xdr-len name-adr name-len -- )
```

```
code# 1 10
```

ATTRIBUTE is the way that properties are passed from an FCode program up to a SunOS device driver. A property consists of two strings: a name string and a value string. The name string gives the name of the property, and the value string gives the value associated with that name. For example, a framebuffer may wish to declare a property named *hres* (for horizontal resolution) with a value of 1152.

The ATTRIBUTE command requires two strings on the stack — the value string and the name string. The name string is an ordinary Forth string, such as any string created with " . The value string, however, *must* be in the xdr format (which stores integers as a 4-byte string, non-null-terminated, and stores all other information as null-terminated strings. See XDRINT, XDR+, XDRINT and XDRSTRING for more information on creating xdr-format strings.

All ATTRIBUTEs created by an FCode program are stored in a "device tree" by the boot PROM. This tree may then be queried by a SunOS device driver, using getprop or getlongprop.

The FCode program and the SunOS device driver may agree on any arbitrary set of names and values to be passed, with virtually no restrictions. Several names, though, have special meaning. For many of them, a shorthand command also exists which makes the attribute declaration a bit simpler.

These special ATTRIBUTES are listed below:

*device_type* display, disk, network. Indicates to the boot PROM that this is a particular type of boot device, i.e. will be used during the booting process. Currently, only display device_types are well-supported. See the command DEVICE-TYPE for more details.

*alias* any two-letter name. Creates a two-letter name for a device, used to specify that device for booting, as in: boot zz()

*name* any . The name attribute is matched by the SunOS identify routine, to match a SunOS device driver with a particular SBus card. Every FCode program *must* declare a name attribute, otherwise SunOS may have difficulty during the booting process. While any string value may be used, Sun recommends

that you use a string of the form `SUNW,xxxxxxxx` . (In place of `SUNW`, use your company's over-the-counter stock symbol. If you're not a publicly-traded company, pick a name that isn't being used.) This technique greatly reduces the chance that the value for your `name` property will accidentally collide with a name chosen by someone else.

*reg* Specifies on-board registers, see `REG` command for more details.

Several other property names, while not restricted, are used by convention. They are:

*intr* Specifies interrupt level, see `INTR` command for more details.

*model* Example: `SUNW,501-1419-01` (model number)

Most `ATTRIBUTE`s are declared during the initial probe-time. Occasionally, however, it will be desirable to declare an `ATTRIBUTE` later, during the `INSTALL` of the device. This would be the case, for example, when declaring the virtual address of a frame buffer, since it is undesirable to memory-map such a large region of memory until it is actually needed. Generally, such a"delayed" `ATTRIBUTE` declaration will be fine, since the boot PROM should always set the device tree pointer back to the correct node before `INSTALL`ing a device.

A bare-minimum FCode program does nothing except to declare a name attribute, as in:

```
FCode-version1
" SUNW,bison-printer" name
end0
```

ATTRIBUTE is head missing in most boot PROMs.

**B#**

```
B# number   ( -- n )
generates:  B(LIT) value
code#       10     xx xx xx xx
```

Interpret the next number in binary (base 2), regardless of any previous settings of HEX, DECIMAL, BINARY or OCTAL. Only the immediately-following number is affected, the current numeric base setting is unchanged. For example:

```
HEX
B# 100(equals decimal 4)
100(equals decimal 256)
```

See also D#, H#, and O#.

**B(")**

```
B(")  ( -- adr len )
code# 12 len xx xx xx ...
```

An internal word, generated by words such as " or ." to leave a text string on the stack. The FCode for B(") should always be followed by an 8-bit length, then by the appropriate number of bytes representing the desired test string. The word B(") should never be used in source code.

**B(')**

```
B(')  ( -- acf )
code# 11 FCode#
```

An internal word, generated by ' or ['] to leave on the stack the code field address of the immediately following word. The FCode for B(') should always be followed by the FCode of the desired word. The word B(') should never be used in source code.

**B(+LOOP)**

```
B(+LOOP)  ( n -- )
code# 16 -offset
```

An internal word, generated by +LOOP. The FCode for B(+LOOP) should always be followed by a negative offset (either 8-bit or 16-bit, see OFFSET16). The word B(+LOOP) should never be used in source code.

**B(:)**

```
B(:)  ( -- )
code# b7
```

An internal word, generated by the defining word : . This is the type entry for : needed by NAMED-TOKEN or NEW-TOKEN. See these words for more details. The word B(:) should never be used in source code.

**B(;)**

```
B(;)  ( -- )
code# c2
```

An internal word, generated by ; to end a colon definition. The word B(;) should never be used in source code.

### B(<MARK)

```
B(<MARK)  ( -- )
code# b1
```

An internal word, generated by `BEGIN`. The word `B(<MARK)` should never be used in source code.

### B(>RESOLVE)

```
B(>RESOLVE)  ( -- )
code# b2
```

An internal word, generated by `REPEAT`, `ELSE`, and `THEN`. The word `B(>RESOLVE)` should never be used in source code.

### B(?DO)

```
B(?DO)  ( end start -- )
code# 18 +offset
```

An internal word, generated by `?DO`. The FCode for `B(?DO)` should always be followed by a positive offset (either 8-bit or 16-bit, see `OFFSET16`). The word `B(?DO)` should never be used in source code.

### B(BUFFER:)

```
B(BUFFER:)  ( n -- )
code# bd
```

An internal word, generated by the defining word `BUFFER:`. This is the type entry for `BUFFER:` needed by `NAMED-TOKEN` or `NEW-TOKEN`. See these words for more details. The word `B(BUFFER:)` should never be used in source code.

### B(CASE)

```
B(CASE)  ( n -- )
code# c4
```

An internal word, generated by CASE . The word B(CASE) should never be used in source code.

### B(CONSTANT)

```
B(CONSTANT)  ( n -- )
code# ba
```

An internal word, generated by the defining word CONSTANT . This is the type entry for CONSTANT needed by NAMED-TOKEN or NEW-TOKEN . See these words for more details. The word B(CONSTANT) should never be used in source code.

### B(CREATE)

```
B(CREATE)  ( -- )
code# bb
```

An internal word, generated by the defining word CREATE. This is the type entry for CREATE needed by NAMED-TOKEN or NEW-TOKEN . See these words for more details. The word B(CREATE) should never be used in source code.

### B(DEFER)

```
B(DEFER)  ( -- )
code# bc
```

An internal word, generated by the defining word DEFER . This is the type entry for DEFER needed by NAMED-TOKEN or NEW-TOKEN . See these words for more details. The word B(DEFER) should never be used in source code.

### B(DO)

```
B(DO) ( end start -- )
code# 17 +offset
```

An internal word, generated by `DO`. The FCode for `B(DO)` should always be followed by a positive offset (either 8-bit or 16-bit, see `OFFSET16`). The word `B(DO)` should never be used in source code.

### B(ENDCASE)

```
B(ENDCASE)  ( -- )
code# c5
```

An internal word, generated by `ENDCASE`. The word `B(ENDCASE)` should never be used in source code.

### B(ENDOF)

```
B(ENDOF)  ( -- )
code# c6
```

An internal word, generated by `ENDOF`. The word `B(ENDOF)` should never be used in source code.

### B(FIELD)

```
B(FIELD)  ( offset size -- offset+size )
code# be
```

An internal word, generated by the defining word `FIELD`. This is the type entry for `FIELD` needed by `NAMED-TOKEN` or `NEW-TOKEN`. See these words for more details. The word `B(FIELD)` should never be used in source code.

### B(IS)

```
B(IS)  ( n -- )
code# c3
```

An internal word, generated by `IS` . The word `B(IS)` should never be used in source code.

### B(LEAVE)

```
B(LEAVE)  ( -- )
code# 1b
```

An internal word, generated by `LEAVE` . The word `B(LEAVE)` should never be used in source code.

### B(LIT)

```
B(LIT)  ( -- n )
code# 10 xx xx xx xx
```

Any input number, such as 205 or -14, will create the `B(LIT)` FCode (code#10), followed by 32-bits (4 bytes) with the actual binary value in two's-complement arithmetic. The number base (hex, decimal or any other chosen radix) is controlled by any previous uses of the tokenizer directives `HEX`, `DECIMAL`, and so on, or by numeric input control words such as `H#`, `D#`, `ASCII`, and so on. Thus,

```
DECIMAL ... 20 ...
```

would be encoded as the hex bytes `10 00 00 00 14`

The only numbers which are not encoded using `B(LIT)` are the values -1, 0, 1, 2, or 3. Because these numbers occur so frequently, these values are assigned individual FCodes to save space.

The word `B(LIT)` should never be used in source code.

### B(LOOP)

```
B(LOOP)  ( n -- )
code# 15 -offset
```

An internal word, generated by `LOOP` . The FCode for `B(LOOP)` shouldalways be followed by a negative offset (either 8-bit or 16-bit, see `OFFSET16`). The word `B(LOOP)` should never be used in source code.

### B(OF)

```
B(OF)  ( testval -- )
code# 1c +offset
```

An internal word, generated by `OF` . The FCode for `B(OF)` should always be followed by a positive offset (either 8-bit or 16-bit, see `OFFSET16`). The word `B(OF)` should never be used in source code.

### B(VALUE)

```
B(VALUE)  ( n -- )
code# b8
```

An internal word, generated by the defining word `VALUE` . This is the type entry for `VALUE` needed by `NAMED-TOKEN` or `NEW-TOKEN` . See these words for more details. The word `B(VALUE)` should never be used in source code.

### B(VARIABLE)

```
B(VARIABLE) ( n -- )
code# b9
```

An internal word, generated by the defining word `VARIABLE` . This is the type entry for `VARIABLE` needed by `NAMED-TOKEN` or `NEW-TOKEN` . See these words for more details. The word `B(VARIABLE)` should never be used in source code.

## B?BRANCH

```
B?BRANCH  ( flag -- )
code# 14 offset
```

An internal word, generated by UNTIL, WHILE, and IF . The FCode for B?BRANCH should always be followed by an offset (either 8-bit or 16-bit, see OFFSET16). The word B?BRANCH should never be used in source code.

## BASE

```
BASE  ( -- adr )
code# a0
```

The address of a VARIABLE containing the current numeric conversion radix to be used when the FCode program is executing, such as 10 for decimal, 16 for hex, 8 for octal, and so on. For example, to print the current value of BASE, use:

```
BASE @  .D
```

Note that BASE @ . would always print "10", since the behavior of . is determined by the value in BASE itself.

The cross-compiler words BINARY, DECIMAL , HEX , or OCTAL are also available for changing the value in BASE as desired. However, these four words behave differently depending whether they occur within a definition or outside of a definition.

If any of BINARY, DECIMAL , HEX , or OCTAL occur *within* a definition, then they will be compiled, later causing a change to the value in BASE when that definition is executed.

If any of BINARY, DECIMAL , HEX , or OCTAL occur *outside* of a definition, however, then they are interpreted as commands to the tokenizer program itself, thus affecting the interpretation of all subsequent numbers in the text.

Note that changes to BASE affect the numeric base of the Toolkit itself, which can create much confusion for any user (the default value for BASE is hexadecimal). If you *must* change the base,

Sun recommends that you save and then restore the original base, as in:

```
: .O  ( n -- )   \ Print n in octal
   BASE @   SWAP    ( oldbase n )
   OCTAL  .         ( oldbase )
   BASE !
;
```

In general, only numeric *output* will be affected by the value in `BASE`. Fixed numbers in FCode source are interpreted by the tokenizer program. Most numeric input is controlled by `BINARY`, `DECIMAL`, `HEX`, `OCTAL`, `B#`, `D#`, `H#`, and `O#`, but these words only affect the tokenizer input base: They but do *not* affect the value in `BASE`. For example:

```
(assume initial value in BASE is 16, i.e. Toolkit is in hex)
(no assumptions should be made about the initial tokenizer base)
FCODE-VERSION1
HEX       (tokenizer in base 16; later execution, using BASE,in base 16)
20 .      (compile decimal 32, later print "20" when FCode executes)
DECIMAL   (tokenizer is in base 10, later execution is in base 16)
20 .      (compile decimal 20, later print "14" since FCode executes in
           hex)
: TEST  ( -- )
   OCTAL   (still compiling in decimal, later change BASE when TEST
            executes)
   20 .    (compiles decimal 20, prints "24" since BASE was just
            changed)
   H# 20  .D  (compiles decimal 32, prints "32"; no permanent base
               changes)
   20 .    (compiles decimal 20, prints "24")
;
20 .     (compile decimal 20, later print "14")
TEST     (prints "24 32 24"; has a side-effect of changing the BASE)
20 .     (compile decimal 20, later print "24" since TEST changed  BASE)

HEX      (tokenizer is in base 16; later execution, using BASE,
          still in base 8)
20 .     (compile decimal 32, later print "40")
```

If this all seems confusing, simply follow these guidelines:

- ❏ *Good*: initially declare HEX just after FCODE-VERSION1 , and make liberal use of B# , D# , O# , H# , . (or .H ) and .D .
- ❏ *Bad*: changing BASE either directly or by calling BINARY , DECIMAL , HEX , or OCTAL from within a definition.

## BBRANCH

```
BBRANCH  ( -- )
code# 13 offset
```

An internal word, generated by AGAIN, REPEAT, and ELSE . The FCode for BBRANCH should always be followed by an offset (either 8-bit or 16-bit,see OFFSET16). The word BBRANCH should never be used in source code.

## BEGIN

```
BEGIN  ( -- )
generates:  B(<MARK)
code# b1
```

Marks the beginning of a conditional loop, such as BEGIN ... UNTIL, BEGIN ... WHILE ... REPEAT, or BEGIN ... AGAIN . See these other words for more details.

## BELL

```
BELL  ( -- n )
code# ab
```

n is the ASCII code for the bell character; decimal 7.

## BETWEEN

```
BETWEEN  ( n min max -- flag )
code# 44
```

`flag` is true if `n` is between `min` and `max`, inclusive of both endpoints ( `min` <= `n` <= `max` ). See `WITHIN` for a different form of comparison.

## BINARY

```
BINARY  ( -- )
code# none  or
generates:  2 BASE !
code#  a7 a0 72
```

If outside of a definition, commands the tokenizer program to interpret subsequent numbers in binary (base 2). If within a definition, change the value in `BASE` affecting later numeric output when the FCode program is executed. See `BASE` .

## BL

```
BL  ( -- n )
code# a9
```

The ASCII code for the `space` character; decimal 32, hex 20 .

## BLANK

```
BLANK  ( adr len -- )
generates:  BL FILL
code# a9 79
```

`len` bytes of memory beginning at `adr` are set to the ASCII character value for space (hex 20). No action is taken if `len` is zero.

## BLJOIN

```
BLJOIN  ( byte.lo byte byte byte.hi -- n )
code# 7f
```

Merge four bytes into a single 32-bit word. Incorrect results will be generated unless the high 24 bits of each byte are zero.

## BODY>

```
BODY>  ( apf -- acf )
code# 85
```

Convert the parameter field address of a word to its code field address. This should not be of interest for most programs.

## >BODY

```
>BODY  ( acf -- apf )
code# 86
```

Convert the code field address of a word to its parameter field address. This should not be of interest for most programs.

## BOUNDS

```
BOUNDS  ( start cnt -- start+cnt start )
code# ac
```

Convert a starting value and count into the form required for a `DO` or `?DO` loop . For example, to perform a loop 20 times, counting up from `4000` to `401F` inclusive, use:

```
4000  20 BOUNDS  DO ... LOOP
```

This is equivalent to:

```
4020 4000  DO ... LOOP
```

## BS

```
BS  ( -- n )
code# aa
```

`n` is the ASCII code for the backspace character; decimal 8.

## BUFFER:

```
BUFFER: name  ( size -- )  at creation
              ( -- adr )   at execution
generates:  new header, B(type) = B(BUFFER:)
code# (header) bd
```

Allocate some memory with `ALLOC-MEM` , and create a name which, when executed, leaves on the stack the virtual address of the desired memory. Create with:

```
200 BUFFER: name
```

See `ALLOC-MEM` for restrictions on available memory.

## BWJOIN

```
BWJOIN  ( byte.lo byte.hi -- word )
code# b0
```

Merge two bytes into the low 16-bits of a stack entry (the upper bits are zero). Incorrect results will be generated unless the high 24 bits of each byte are zero.

## 4-BYTE-ID

```
4-BYTE-ID  ( -- )
code# fe
```

This byte (at location 0) followed by 3 more identifier bytes, was used during some of the early boot PROM development as a replacement for actual FCode, by providing a single "magic" number to identify an SBus device. It was a temporary measure only, as it required the boot PROM to "know" the correct magic number for a given device.

This feature is no longer supported, and should not be used under any circumstances.

## C!

```
C!  ( n adr -- )
code# 75
```

The least significant 8 bits of `n` are stored in the byte at `adr` .

## C,

```
C,  ( n -- )
code# d0
```

Compile a byte into the dictionary. This word may be used, in conjunction with `CREATE` , to create an array-type structure, as:

```
CREATE name   77 c,  23 c,  ff c,  ff c,  47 c,  22 c,  ...
```

Later execution of *name* leaves the address of the first byte of the array (the address of the byte '77') on the stack.

`C,` should be used within definitions only with extreme caution. Any such usage has the potential (if the definition were to be called repeatedly) to waste large amounts of run-time dictionary space, which is currently in limited supply.

## C@

```
C@  ( adr -- n )
code# 71
```

The byte at address `adr` is placed into the low 8-bits of `n` (the upper bits are padded with zeroes).

### /C

```
/C  ( -- n )
code# 5a
```

`n` is the size in bytes of a byte, which is 1. See `/W`, `/L`, and `/N`.

### /C*

```
/C* ( n1 -- n2 )
code# 66
```

`n2` is the result of multiplying `n1` by the length in bytes of a byte. This is useful for converting an index into a byte offset. On a byte-addressed machine such as this one, this doesn't do anything.

### CA+

```
CA+  ( adr1 index -- adr2 )
code# 5e
```

`adr2` is the address of the `index`'th character after `adr1`. For byte-addressed machines (such as this one), this is equivalent to `+`. `CA+` should be used in preference to `+` when calculating addresses because it more clearly expresses the intent of the operation and is more portable.

### CA1+

```
CA1+  ( adr1 -- adr2 )
code# 62
```

`adr2` is the address of the next byte after `adr1` . For byte-addressed machines (such as this one), this is equivalent to `1+` . `CA1+` should be used in preference to `1+` because it more clearly expresses the intent of the operation and is more portable.

## CARRET

```
CARRET  ( -- n )
generates:  B(LIT) 13(decimal)
code# 10 00 00 00 0d
```

`n` is the ASCII code for the carriage return character; decimal 13, hex `0d`.

## CASE

```
CASE  ( n -- )
generates:  B(CASE)
code# c4
```

A case statement is started which selects its action based on the value of selector. Example of use:

```
: FOO ( selector -- )
   CASE
     0  OF  ." It was 0"   ENDOF
     5  OF  ." It was 1"   ENDOF
    -2  OF  ." It was 2"   ENDOF
    ( selector) ." It was " DUP U.  \ Default clause.
  ENDCASE
;
```

The default clause is optional. When an `OF` clause is executed, the selector is *not* on the stack. When a default clause is executed, the selector *is* on the stack. The default clause may use the selector, but must not remove it from the stack (it will be automatically removed just before the `endcase`). At run time, `OF` tests the top of the stack against the selector. If they are the same, the selector is dropped and the following Forth code is executed. If they are not the same, execution continues at the point just following the matching `ENDOF` .

## CMOVE

```
CMOVE  ( adr1 adr2 len -- )
generates:  MOVE
code# 78
```

Copy `len` bytes of an array starting at `adr1` to `adr2` . This word simply calls `MOVE` , which is "smart" and handles overlapping arrays in either direction correctly.

`CMOVE` and `CMOVE>` are older standard Forth words which explicitly command in which order to copy the bytes (back-to-front, or front-to-back). In most cases, the distinction is not important. This distinction is important if the arrays overlap, else the source array may be overwritten prematurely, with unexpected results.

**Note**: Code using `CMOVE` or `CMOVE>` , which depends on this "interesting" result will have to be rewritten.

Also, `MOVE` will perform 16-bit, 32-bit or possibly even 64-bit operations (for better performance) if the alignment of the operands permit. If your hardware requires explicit 8-bit or 16-bit accesses, you will probably wish to use an explicitly-coded `DO ... LOOP` instead.

## CMOVE>

```
CMOVE>  ( adr1 adr2 len -- )
generates:  MOVE
code# 78
```

Copy `len` bytes of an array starting at `adr1` to `adr2` . This word simply calls `MOVE` . See `CMOVE` for more information.

## COMP

```
COMP  ( adr1 adr2 len -- n )
code# 7a
```

Compare two byte arrays starting at addresses `adr1` and `adr2` and continuing for `len` bytes. `n` is 0 if the arrays are the same.

`n` is 1 if the first differing character in the array at `adr1` is numerically greater than the corresponding character in the array at `adr2` . `n` is -1 if the first differing character in the array at `adr1` is numerically less than the corresponding character in the array at `adr2` .

## CONSTANT

```
CONSTANT name  ( n1 -- ) at creation
               ( -- n1 ) at execution
generates:  new header, B(type) = B(CONSTANT)
code# (header) ba
```

Creates a named constant. The name is initially created with:

```
456 CONSTANT name
```

where the number before `CONSTANT` is the desired value for `name` . Later occurrences of `name` will leave the correct value on the stack. `CONSTANT` values should never be changed by the program. If you wish to change the value of a `CONSTANT` by the program, you should declare it to be a `VALUE` instead.

## CONTROL

```
CONTROL x  ( -- n )
generates:  B(LIT) value
code# 10 00 00 00 xx
```

Interpret the next letter as a control-code. For example:

```
CONTROL c    (equals 03)
```

## COUNT

```
COUNT  ( pstr -- adr len )
code# 84
```

Convert a packed string into a byte-array format. `pstr` is the address of a packed string, where the byte at address `pstr` is the length of the string and the string itself starts at address `pstr+1` .

Packed strings are generally not used in FCode. Virtually all string operations are in the "`adr len`" format.

## CR

```
CR  ( -- )
code# 92
```

A `DEFER` word which is used to terminate the line on the display and go to the next line. The default implementation transmits a carriage return and line feed to the display, clears `#OUT` , and adds 1 to `#LINE` .

`CR` should be used whenever you wish to start a new line of output, or to force any previously buffered output text to be displayed. This forcing is valuable for outputting error messages, to ensure that the error message is sent *before* any system crash.

## (CR

```
(CR ( -- )
code# 91
```

Output the carriage return character (CARRET, hex 0d). This word is not commonly used, see CR .

## CREATE

```
CREATE name  ( -- )      at creation
             ( -- adr )  at execution
generates:  new header, B(type) = B(CREATE)
code# (header) bb
```

Create a name. At execution time it returns the address of memory immediately following the name in the dictionary. This may be used to create an array-type structure, as:

```
CREATE name   77 c,  23 c,  ff c,  ff c,  47 c,  22 c,  ...
```

Later execution of name leaves the address of the first byte of the array (here, the address of the byte '77') on the stack.

In the current implementation, CREATE may *not* be used within definitions in an FCode program. The common Forth constructs CREATE...DOES> are not supported.

## .D

```
.D  ( n -- )
generates:  BASE @  SWAP DECIMAL                  .  BASE !
generates:  BASE @  SWAP D# 10           BASE !   .  BASE !
code#       a0   6d 49   10 00 00 00 0a a0   72 9d  a0   72
```

n is displayed in decimal (using . ) The value of BASE is not permanently affected.

## D#

```
D# number( -- n )
generates:  B(LIT) value
code# 10 value
```

Interpret the next number in decimal (base 10), regardless of anyprevious settings of HEX , DECIMAL , BINARY, or OCTAL . Only the immediately following number is affected, the default numeric base setting is unchanged. For example:

```
HEX
D# 100  (equals decimal 100)
100     (equals decimal 256)
```

See also B# , H# , and O# .

## DECIMAL

```
DECIMAL   ( -- )
code# none  -or-
generates: B(LIT) 10(decimal) BASE !
code#      10     00 00 00 0a a0   72
```

If outside of a definition, commands the tokenizer program to interpret subsequent numbers in decimal (base 10). If within a definition, change the value in BASE affecting later numeric output when the FCode program is executed. See BASE .

## DEFER

```
DEFER name  ( -- )          at creation
            ( ?? -- ?? )    at execution
generates:  new header, B(type) = B(DEFER)
code# (header) bc
```

Create a DEFER'd executable. This is a word which has a variable behavior, depending on the function which is later loaded into it. The name is initially created with:

```
DEFER name
```

Later, after some other word *foobar* has been created, this behavior can then be loaded in, with:

```
['] foobar  IS  name
```

DEFER'd words are useful for generating recursive routines. Here's an example:

```
DEFER hold2  \ Will execute action2
: action1
  ...
  hold2  ( really action2 )
  ...  ;
: action2
  ...
  action1
  ...  ;
' action2  IS  hold2
```

DEFER'd words can also be used for creating words with different behaviors depending on your needs. For example:

```
DEFER .special ( n -- ) \ Print a value, using special techniques
: print-em-all  ( -- )
  ...  .special
  ...  .special
  ...  .special
;

( .D prints in decimal )
( .H prints in hex )
( .SP prints in a custom format )
: print-all-styles
  ['] .D  IS .special   print-em-all
  ['] .H  IS .special   print-em-all
  ['] .SP IS .special   print-em-all
;
```

If a DEFER word is executed before being loaded with some behavior, an error message will be printed.

## DEPTH

```
DEPTH  ( -- +n )
code# 51
```

+n is the number of entries contained in the data stack, not counting itself. Note that when an FCode program is called, there could be other items on the stack from the calling program.

DEPTH is especially useful for before/after stack depth checking, to determine if the stack was corrupted by a particular operation.

## DEVICE-TYPE

```
DEVICE-TYPE  ( adr len -- )
code# 1 1a
```

This is a shorthand word for creating a "device_type" property. This property is essential for any SBus device which will be used during booting, as it tells the boot PROM which type of boot device it is. An example usage would be:

```
" display"  device-type
```

This is exactly equivalent to the following:

```
" display"  xdrstring  " device_type"  attribute
```

Note the spelling difference between the FCode command DEVICE-TYPE (hyphen) and the device_type property (underscore). The device_type property is passed to the SunOS device driver as usual, but is also looked at and used by the boot PROM as well.

The key values used by the boot PROM are:

- *display* - for a display device. The boot PROM expects this device to implement the terminal emulation FCode primitives described elsewhere, and `IS-INSTALL`, `IS-REMOVE`, and `IS-SELFTEST`.
- *network* - for a network interface. This feature is not yet fully supported.
- *disk* - for a disk interface. This feature is not yet fully supported.

Other key values will be added as future device support is implemented.

When the boot PROM needs a device (such as a framebuffer) after probing is completed, it searches its internal list (created during probing) for a device-type of the appropriate form. If more than one such device exists, the boot PROM chooses one somehow. This is currently done by choosing the first one probed, but this could change in the future. This word is head missing in most boot PROMs.

## DIAGNOSTIC-MODE?

```
DIAGNOSTIC-MODE?  ( -- flag )
code# 1 20
```

Returns a `TRUE` flag if the `diag-switch?` NVRAM parameter is set to `TRUE`. This word enables an FCode program to optionally perform some extended selftests, based on the `diag-switch?`. For example:

```
diagnostic-mode?
if    do-extended-tests
else  do-normal-tests
then
```

This word is head missing in most boot PROMs.

## DIGIT

```
DIGIT  ( char base -- digit true | char false )
code# a3
```

If the character `char` is a digit in the specified base, returns the numeric value of that digit under `true` , else returns the character under `false` . Appropriate characters are hex `30-39` (for digits 0-9) and hex `61-66` (for digits `a-f`), depending on base.

## DISPLAY-STATUS

```
DISPLAY-STATUS   ( n -- )
code# 1 21
```

Display the results of some test. The method of display is system-dependent. On some systems, a bank of 8 LEDs is used. On the SPARCstation 1 and 1+, if `n` is -1 (`TRUE`) then the front-panel LED will be turned off, otherwise it will be turned on. This word is head missing in most boot PROMs.

## DO

```
DO  ( limit start -- )
generates:  B(DO) +offset
code# 17 +offset
```

Begin a counted loop in the form `DO ... LOOP` or `DO ... +LOOP` .The loop index begins at `start` , and terminates based on `limit` . See `LOOP` and `+LOOP` for details on how the loop is terminated. The loop is always executed at least once. For example:

```
8 3  DO  I .  LOOP      \ would print  3 4 5 6 7
9 3  DO  I .  2 +LOOP   \ would print  3 5 7
```

## ?DO

```
?DO  ( limit start -- )
generates:  B(?DO) +offset
code# 18 +offset
```

Begin a counted loop in the form `?DO ... LOOP` or `?DO ... +LOOP` . The loop index begins at `start` , and terminates based on `limit` . See `LOOP` and `+LOOP` for details on how the loop is terminated. Unlike `DO` , if `start` is equal to `limit` the loop is executed zero times. For example:

```
8 1  ?DO  I .  LOOP     \   would print  1 2 3 4 5 6 7
2 1  ?DO  I .  LOOP     \   would print  1
1 1  ?DO  I .  LOOP     \   would print nothing
1 1   DO  I .  LOOP     \   would print  1 2 3 4 5 6 7 8 9
                                     \   ... (all numbers)
```

`?DO` may be used in place of `DO` in nearly all circumstances.

## DMA-ALLOC

```
DMA-ALLOC  ( #bytes -- virtual )
code# 1 01
```

Used to allocate some memory for DMA purposes. The allocated memory may be returned to the system with `FREE-VIRTUAL` .

Because of the way that memory is allocated in some early boot PROMs, it is best to allocate all of your memory in one operation and then to subdivide it according to your needs.

This word is head missing in some early boot PROMs.

## DRIVER

```
DRIVER  ( adr len -- )
code# 1 18
```

This is a shorthand word for creating a `name` property. It strips off the first 5 characters from the given string, and creates a `name` property from the remainder of the string. An example usage would be:

```
" SUNW,liontamer"  driver
```

This is exactly equivalent to the following:

```
" liontamer"  xdrstring  " name"  attribute
```

The original idea was that the `SUNW` would be stored in a separate "manufacturer" property, but this was never implemented. Recall that the name property is matched by the SunOS "identify" routine, to match a SunOS device driver with a particular SBus card. Every FCode program *must* include a `name` property declaration.

Sun recommends that you don't use the `DRIVER` command in future SBus designs. It has two major problems:

1) It always strips off exactly 5 characters, instead of more properly stripping `n` characters up to the ",".
2) It creates a `name` attribute without any manufacturer keywords attached.

This will eventually create name conflicts between different manufacturers. See `ATTRIBUTE` for the proper method for creating the `name` property. `DRIVER` will continue to be supported, because there are already several known FCode drivers which call it. But again, Sun strongly recommends that `DRIVER` *not* be used for any current or future SBus devices. You should declare the name property explicitly, instead. This word is head missing in most boot PROMs. See also `NAME` and `ATTRIBUTE` .

### DROP

```
DROP  ( n -- )
code# 46
```

Removes one item from the stack.

### 2DROP

```
2DROP  ( n1 n2 -- )
code# 52
```

Removes two items from the stack.

### DUP

```
DUP  ( n1 -- n1 n1 )
code# 47
```

Duplicates the top stack item.

### ?DUP

```
?DUP  ( n1 -- 0 | n1 n1 )
code# 50
```

Duplicate the top stack item unless it is zero.

### 2DUP

```
2DUP  ( n1 n2 -- n1 n2 n1 n2 )
code# 53
```

Duplicates the top two stack items.

### 3DUP

```
3DUP  ( n1 n2 n3 -- n1 n2 n3 n1 n2 n3 )
```

```
generates:  2 PICK 2 PICK 2 PICK
code# a7 4e a7 4e a7 4e
```

Duplicates the top three stack items.

## ELSE

```
ELSE  ( -- )
generates:  BBRANCH +offset B(>RESOLVE)
code# 13 +offset b2
```

Begin the `ELSE` clause of an `IF ... ELSE ... THEN` statement. See `IF` for more details.

## EMIT

```
EMIT  ( char -- )
code# 8f
```

A `DEFER` word that outputs the indicated ASCII character. For example, (hex) `41 EMIT` outputs an "A", `62 EMIT` outputs a "b", `34 EMIT` outputs a 4.

## EMIT-BYTE

```
EMIT-BYTE  ( n -- )
code# n
```

This is a tokenizer command, used to manually output a desired byte of FCode. Only use it together with `TOKENIZER[` .

This command would be useful, for example, if you wished to generate a new FCode command that the tokenizer did not understand. This command should be

used with caution or else an invalid FCode program will result.

### END0

```
END0 ( -- )
code# 00
```

A word which marks the end of an FCode program. This word must be present at the end of your program, or erroneous results may occur.

### END1

```
END1  ( -- )
code# ff
```

An alternate word for `END0` , to mark the end of an FCode program. `END0` is recommended.

### ENDCASE

```
ENDCASE  ( -- )
generates:  B(ENDCASE)
code# c5
```

Marks the end of a `CASE` statement. See `CASE` for more details.

### ENDOF

```
ENDOF  ( -- )
generates:  B(ENDOF) +offset
code# c6 +offset
```

Marks the end of an `OF` clause within a `CASE` statement. See `CASE` for more details.

## ERASE

```
ERASE  ( adr len -- )
generates:  0 FILL
code# a5 79
```

Sets `len` bytes of memory beginning at `adr` to zero. No action is taken if `len` is zero.

## EXECUTE

```
EXECUTE  ( acf -- )
code# 1d
```

Executes the word definition whose compilation address is `acf` . An error condition exists if `acf` is not a compilation address.

## EXIT

```
EXIT  ( -- )
code# 33
```

Compiled within a colon definition. When encountered, execution leaves the current word and renturns control to the calling word. May not be used within a `DO` loop.

## EXPECT

```
EXPECT  ( adr len -- )
code# 8a
```

A `DEFER` word that receives a line of characters from the keyboard and stores them into memory, performing line editing as the characters are typed. Displays all characters actually received and stored into memory. The number of received characters is stored in `SPAN` .

The transfer begins at `adr` proceeding towards higher addresses one byte per character until either a `return` is received or until `len` characters have been transferred. No more than `len`

characters will be stored. The `return` is not stored into memory. No characters are received or transferred if `len` is zero.

## FALSE

```
FALSE  ( -- 0 )
generates:  0
code# a5
```

Leave the value for the `FALSE` flag (which is zero) on the stack.

## FCODE-VERSION1

```
FCODE-VERSION1  ( -- )
generates:  VERSION1 (null) (checksum) (length)
code#       fd 00            xx yy      aa bb cc dd
```

This word *must* be the first command in your FCode program (except for tokenizer directives such as `HEX` or `\` which do not generate any FCode bytes). The command `FCODE-VERSION1` creates an 8-byte header, as:

```
(fd) VERSION1                    (1 byte)
(00) Null byte, always 0         (1 byte)
(xxyy) Checksum                  (2 bytes)
(aabbccdd) Length                (4 bytes)
```

Calculates the checksum by summing all remaining bytes, from just after the length field to the end of the usable FCode data, as indicated by the `length` field.

The `length` field specifies the total usable length of FCode data, from `VERSION1` to `END0` inclusive. Additional `END0` bytes are appended to the end of the data, if needed, to leave a total length which is evenly divisible by 4. The "null byte" position may be used in the future to carry a version number or other information, but it is currently not used.

## FIELD

```
FIELD  ( offset size -- offset+size )  at creation
       ( base -- base+offset )         at execution
generates:  new header, B(type) = B(FIELD)
code# (header) be
```

STRUCT and FIELD are used to create named offset pointers into some array structure. For each field in the array structure, a name is assigned to the location of that field (as an offset from the beginning of the array). Here's a code example. (The numbers in parentheses show the stack *after* each word is created.) The structure being described is:

```
Byte#
0 1  2 3 4 5  6    7   8 9      10 11 12 13 14 15 16 17  18 19
size flags..  bits key fullname.......................  age
                     initials lastname..............

STRUCT            ( 0 )
2 FIELD size      ( 2 )     \ Equiv. to:   : size      0 + ;
4 FIELD flags     ( 6 )     \ Equiv. to:   : flags     2 + ;
1 FIELD bits      ( 7 )     \ Equiv. to:   : bits      6 + ;
1 FIELD key       ( 8 )     \ Equiv. to:   : key       7 + ;
0 FIELD fullname  ( 8 )     \ Equiv. to:   : fullname  8 + ;
2 FIELD initials  ( 10 )    \ Equiv. to:   : initials  8 + ;
8 FIELD lastname  ( 18 )    \ Equiv. to:   : lastname 10 + ;
2 FIELD age       ( 20 )    \ Equiv. to:   : age      18 + ;
  CONSTANT /record ( -- )   \ Equiv. to:   20 CONSTANT /record
```

Typical usage of these defined words would be as:

```
/record BUFFER: myrecord   \ Create the "myrecord" buffer
myrecord  flags L@  \ get flags data
myrecord  key   C@  \ get key data
myrecord  size  W@  \ get size data
/record             \ get total size of the array
```

Note that STRUCT is merely a cross-compiler equivalent which puts the number 0 on the stack.

## FILL

```
FILL  ( adr u byte -- )
code# 79
```

Set `u` bytes of memory beginning at `adr` to `byte` . No action taken if `u` = 0.

## $FIND

```
$FIND  ( adr len -- adr len false | acf +/-1 )
code# cb
```

Takes a string from the stack, and tries to find that word in the Open Boot PROM. This is an escape hatch, allowing an FCode program to perform any function which is available in the Open Boot PROM Toolkit but which is not defined as part of the standard FCode interface.

If the word is not found, the original string is left on the stack, with a `FALSE` on top of the stack. If the word is found, the code field address of that word is left on the stack, and either a +1 or -1 is left on top. +1 is left if the found word is an `IMMEDIATE` word, -1 is left otherwise.

Use `$FIND` with caution! Different CPUs or even different versions of the boot PROM may change or delete certain words in the Toolkit. If your FCode program depends on one of these words, you may suddenly find that your SBus card doesn't work properly with future releases.

**Caution**: If you find yourself tempted to use `$FIND` , please contact the Sun SBus Support Group and tell them what function you need to use this way. This will help Sun to plan for future FCode features, and will let you know the likelihood of the needed Toolkit word being changed in the future.

Example of use:

```
" root-info" $FIND   ( adr len false | acf +/-1 )
IF  EXECUTE       \ If found, then do the function
                  \ In this example, we don't care about
                  \ immediate vs. non-immediate
ELSE  ( adr len )   TYPE   ."  was not found!" CR
THEN
```

`$FIND` is head missing in many boot PROMs.

## FINISH-DEVICE

```
FINISH-DEVICE  ( -- )
code# 1 27
```

The two words `FINISH-DEVICE` and `NEW-DEVICE` let a single FCode program declare more than one entry into the device tree. This capability is useful when a single SBus card contains two or more essentially independent devices, to be controlled by two or more separate SunOS device drivers. Typical usage:

```
FCODE-VERSION1
...driver#1...
FINISH-DEVICE     \ Terminate device tree entry#1
NEW-DEVICE        \ Begin a new device tree entry
...driver#2
FINISH-DEVICE     \ Terminate device tree entry#2
NEW-DEVICE        \ Begin a new device tree entry
...driver#3...
end0
```

There is an implicit `NEW-DEVICE` call at the beginning of an FCode program (at `FCODE-VERSION1`), and an implicit `FINISH-DEVICE` call at the end of an FCode program (at `END0`). Thus, FCode programs which only define a single device and driver will never need to call `FINISH-DEVICE` or `NEW-DEVICE` . `FINISH-DEVICE` is not defined in SPARCstation 1 boot PROM versions 1.0 and 1.1 .

If your application requires multiple devices per SBus card, you will need to require the user to upgrade their system to a 1.2 or

greater boot PROM. This word is head missing in most boot PROMs.

## FLIP

```
FLIP  ( n1 -- n2 )
code# 80
```

`n2` is the result of exchanging the two low-order bytes of the number `n1` . The two upper bytes of `n1` must be zero, or erroneous results will occur.

## FLOAD

```
FLOAD  filename  ( -- )
code# none
```

Tokenizer command that begins tokenizing text in the name file. When the named file is done, tokenizing continues on the file that called `filename` with `FLOAD`.

`FLOAD` acts like an `#INCLUDE` statement. `FLOAD` commands may be nested; an `FLOAD`ed file may include `FLOAD` commands.

`FLOAD` is useful for creating large FCode programs, making it easier to break them up into function blocks for better clarity and portability.

**Note**: `FLOAD` commands won't work when downloading text in source-code form. You can either manually merge your text into one big file, or else download and execute the various file separately.

## FREE-MEM

```
FREE-MEM  ( virtual #bytes -- )
code# 8c
```

Frees up memory allocated by `ALLOC-MEM`. Currently, there is an order restriction for using `FREE-MEM`. Memory must be freed inthe reverse order to which it was allocated (like using a stack).

## FREE-VIRTUAL

```
FREE-VIRTUAL  ( virtual size -- )
code# 1 05
```

Undoes the MMU page map entries generated by `MEMMAP`, `DMA-ALLOC`, or `MAP-SBUS`: it unmaps the given address.

Mappings should always be unmapped when no longer needed. In particular, any mappings performed (for example, for some power-on initialization) during the initial SBus card probe (when the FCode is first interpreted) must be unmapped during probe as well.

If you don't do this, the boot PROM may well do it for you. Also, any mappings performed during `install` should be unmapped during `remove` (see `IS-INSTALL` and `IS-REMOVE`).

## GET-MSECS

```
GET-MSECS  ( -- ms )
code# 1 25
```

Returns the current value in a free-running system counter. The number returned is a running total, expressed in milliseconds. You can use this for measuring time intervals (by comparing the starting value with the ending value). No assumptions should be made regarding the absolute number returned; only relative interval comparisons are valid.

No assumptions should be made regarding the *precision* of the number returned. In many systems (including the SPARCstation 1), the value is derived from the system clock,

which typically ticks once per second. Thus, the value returned by GET-MSECS on the SPARCstation 1 and 1+ will be seen to increase in jumps of 1000 (decimal), once per second. For a delay timer of millisecond accuracy, see MS .

## GROUP-CODE

```
GROUP-CODE  ( -- adr )
code# 1 23
```

A VARIABLE containing a group offset for distinguishing various selftests in MEMORY-TEST-SUITE . The value in GROUP-CODE is added to the code# for individual tests, and the sum is displayed with DISPLAY-STATUS . Normally, GROUP-CODE should be set to 0, using the command:

```
0 group-code !
```

GROUP-CODE affects the reporting of errors through the LEDs from MEMORY-TEST-SUITE . It is the high-order 4 bits of the number displayed on the LEDs. The lower 4 bits encode the individual test number.

This word is head missing in most boot PROMs.

## .H

```
.H  ( n -- )
generates:  BASE @ SWAP  HEX                  . BASE !
generates:  BASE @ SWAP  D# 16        BASE !  . BASE !
code#       a0   6d 49   10 00 00 00 10 a0 72 9d a0 72
```

Displays n in hex (using . ) The value of BASE is not permanently affected.

## H#

```
H# number  ( -- n )
generates:  B(LIT) value
code# 10 xx xx xx xx
```

Interpret the next number in hex (base 16), regardless of any previous settings of `HEX` , `DECIMAL` , `BINARY` , or `OCTAL`. Only the immediately following number is affected, the default numeric base setting is unchanged. For example:

```
DECIMAL
H# 100  (equals decimal 256)
100     (equals decimal 100)
```

See also `B#` , `D#` , and `O#` .

## HEADERLESS

```
HEADERLESS  ( -- )
code# none
```

Causes subsequent definitions to be created in FCode without the name field (the "head"). (See `NAMED-TOKEN` and `NEW-TOKEN` .) This is sometimes done to save space in the final FCode PROM, or possibly to make it more difficult to reverse-engineer an FCode program.

All such headerless words may be used normally within the FCode program, but cannot be called interactively from the Toolkit for testing and development purposes.

Unless PROM space and/or dictionary space is a major consideration, Sun recommends not using `HEADERLESS` words, because they make debugging more difficult.

## HEADERS

```
HEADERS  ( -- )
code# none
```

Cancels a previous HEADERLESS command, so that subsequent definitions will be saved with the name field (the "head") intact. This is the initial default behavior.

Note that even normal FCode words (with heads) cannot be called interactively from the Toolkit unless the NVRAM parameter FCODE-DEBUG? has been set to TRUE before a system reset.

## HERE

```
HERE  ( -- adr )
code# ad
```

adr is the address of the next available dictionary location.

## HEX

```
HEX  ( -- ) -or-
generates: B(LIT) 16(decimal)  BASE !
code#      10     00 00 00 10  a0   72
```

If outside of a definition, commands the tokenizer program to interpret subsequent numbers in hex (base 16). If within a definition, change the value in BASE affecting later numeric output when the FCode program is executed. See BASE .

## HOLD

```
HOLD  ( char -- )
code# 95
```

`Inserts char` into a pictured numeric output string. Typically used between `<#` and `#>` .

## I

```
I  ( -- n )
code# 19
```

`n` is a copy of the loop index. May only be used inside of a `DO` or `?DO` loop.

## IF

```
IF  ( flag -- )
generates:  B?BRANCH +offset
code# 14 +offset
```

Execute the following code if `flag` is true. Used in the form:

```
flag  IF ... ELSE ... THEN          or
flag  IF ... THEN
```

If `flag` is true, the words following `IF` are executed and the words following `ELSE` are skipped. The `ELSE` part is optional. If `flag` is false, words from `IF` through `ELSE` , or from `IF` through `THEN` (when no `ELSE` is used), are skipped.

## INTR

```
INTR  ( SBus-intr-level  vector -- )
code# 1 17
```

This is a shorthand word for declaring an `ATTRIBUTE` named `intr`. (By convention, `intr` is used for declaring the interrupt level for an SBus device.) Typical usage:

```
5  0  intr
```

This declares an interrupt level for the device of SBus interrupt level 5. The interrupt vector is null. The following code would accomplish exactly the same thing:

```
5  sbus-intr>cpu  xdrint
0                 xdrint  xdr+
" intr" attribute
```

Translates the SBus interrupt specified (values 1-7) into an appropriate CPU-level interrupt by `SBUS-INTR>CPU`, and then passes the CPU interrupt level to the SunOS device driver by the `intr` property.

This technique assures that SBus cards will work properly on future CPU architectures which may have a different interrupt structure. Note that if you need to declare more than one interrupt level, you *must* use the longer, more explicit method in order to build the structure to be passed into the `intr` property. For example, to declare SBus interrupt levels 3 and 5, use:

```
3  sbus-intr>cpu  xdrint         \ Interrupt#1
0                 xdrint  xdr+   \ Null vector#1
5  sbus-intr>cpu  xdrint  xdr+   \ Interrupt#2
0                 xdrint  xdr+   \ Null vector#2
" intr" attribute
```

The table below outlines the SBus to SPARCstation interrupt mappings. Note that a different mapping may well be used for future architectures, so your FCode must *not* depend on this knowledge! Use this mapping in order to select an appropriate

SBus interrupt level. An approximately similar CPU interrupt level will always be mapped to the SBus on future architectures.

| SBus | SPARCstation 1 | Usage |
|---|---|---|
| | 15 | Asynchronous Memory Error |
| | 14 | Counter 1 |
| | 13 | Audio |
| | 12 | Keyboard,Mouse,Serial Ports |
| | 11 | Floppy |
| | 10 | Counter 0 |
| 7 | 9 | |
| 6 | 8 | |
| 5 | 7 | Video |
| | 6 | Software request 6 |
| 4 | 5 | Ethernet |
| | 4 | Software request 4 |
| 3 | 3 | SCSI, DMA |
| 2 | 2 | |
| 1 | 1 | Software request 1 |

The 1.0 and 1.1 boot PROMs for SPARCstation 1 incorrectly mapped SBus interrupts 6 and 7 to 9 and 13 instead of 8 and 9, so the value passed with the `intr` property will be incorrect. If you need SBus interrupt #6 or #7, you'll need to either require the user to install a 1.2 or later boot PROM, or you'll have to adjust for this error in your SunOS driver. You can get the PROM version number from `(*romp->v_mon_id)` for example: `00 01 00 03` (hex) for a boot PROM 1.3

The `vector` parameter with `INTR` is typically 0.

This word is head missing in most boot PROMs. See also `ATTRIBUTE`.

## IS

```
IS name  ( n -- )
generates:  B(IS) FCode
code# c3 FCode
```

Changes the contents of a `VALUE` or a `DEFER` word:

```
number   IS name    (for a VALUE)
acf      IS name    (for a DEFER word)
```

## IS-INSTALL

```
IS-INSTALL  ( acf -- )
code# 1 1c
```

For any SBus device which is to be used by the boot PROM beforeor during booting, `IS-INSTALL` declares the FCode procedure which should be used to install (i.e. initialize) that device. Note that this is distinct from any once-only power-on initialization, which should be performed during the probing process itself.

The `IS-INSTALL` routine and `IS-REMOVE` routine should comprise a matched pair, which may be performed alternately as many times as needed. Typically, the `IS-INSTALL` routine performs mapping functions and some initialization, and the `IS-REMOVE` performs any cleanup functions and then does a complementary unmapping.

Here is some partial, typical code:

```
FCode-version1
...
: power-on   ( -- )   \ Once-only, power-on initialization
   map-register
   init-register
   unmap-register
;
...
```

```
: map-devices  ( -- )   \ Map register and buffer
   map-register
   map-buffer
;
...
: install-me   ( -- )   \ Do this to start using this device
   map-devices
   initialize-devices
;
: remove-me    ( -- )    \ Do this to stop using this device
   reset-buffers
   unmap-devices
;
...
\ This routine executed during the "probe" of this FCode
: my-probe  ( -- )     \ First, define the routine
   power-on                    \ Power-on initialization
   ['] install-me  IS-INSTALL   \ Declare "install" routine
   ['] remove-me   IS-REMOVE    \ Declare "remove" routine
   ['] test-me     IS-SELFTEST  \ Declare "selftest" routine
;                      \ End of the definition
my-probe               \ Now, actually execute this routine
END0
```

This word is head missing in most boot PROMs.

## IS-REMOVE

```
IS-REMOVE  ( acf -- )
code# 1 1d
```

Declares the routine which will deallocate a device that is no longer going to be used. Typical deallocation would include unmapping memory and clearing buffers. Usage:

```
['] my-remove-routine  IS-REMOVE
```

The routine loaded with `IS-REMOVE` should form a matched pair with the routine loaded with `IS-INSTALL` . See `IS-INSTALL` for more details. This word is head missing in most boot PROMs.

## IS-SELFTEST

```
IS-SELFTEST   ( acf -- )
code# 1 1e
```

Declares the routine which will perform a self test of the device. Usage:

```
['] my-selftest-routine  IS-SELFTEST
```

This declaration is typically performed in the same place in the code as `IS-INSTALL` and `IS-REMOVE` .

The self test routine should return a status parameter on the stack indicating the results of the test. A zero value indicates that the test passed. Any non-zero value indicates that the self test failed, but the actual meaning for any non-zero value is not specified. ( `MEMORY-TEST-SUITE` returns a flag meeting these specifications.)

Self test is not automatically executed. In future PROMs, Sun will provide a `test` command to interactively execute the self test entry.

For automatic testing, devices should perform a quick sanity check as part of the `INSTALL` routine. This word is head missing in most boot PROMs.

## J

```
J  ( -- n )
code# 1a
```

`n` is a copy of the index of the next outer loop. May only be used within a nested `DO` or `?DO` loop. For example:

```
DO
  ...
    DO ... J ... LOOP
  ...
LOOP
```

Usually, DO loops should not be nested this deeply inside a single definition. Forth programs are generally more readable if inner loops are defined inside a separate word.

## KEY

```
KEY  ( -- char )
code# 8e
```

A DEFER word that reads the next ASCII character from the keyboard. If no character has been typed since KEY was last executed, KEY waits until a new character is typed. All valid ASCII characters can be received. Control characters are not processed by the system for any editing purpose. Characters received by KEY are not displayed.

## KEY?

```
KEY?  ( -- flag )
code# 8d
```

A DEFER word returning true if a character has been typed on the keyboard since the last time that KEY or EXPECT was executed. The keyboard character is not consumed.

Use KEY? to make simple, interruptable infinite loops:

```
BEGIN   ...   KEY? UNTIL
```

The contents of the loop will repeat indefinitely until any key is pressed.

**L!**

```
L!  ( n adr -- )
code# 73
```

The 32-bit value `n` is stored at location `adr` (through `adr+3` ). The highest byte is stored at `adr` ; the lowest byte is stored at `adr+3` . `adr` must be on a 32-bit boundary; it must be evenly divisible by 4.

**L,**

```
L,  ( n -- )
code# d2
```

Compile 4-bytes into the dictionary, starting with the highest byte. See `C,` for limitations.

**L@**

```
L@  ( adr -- n )
code# 6e
```

Fetch the 32-bit number stored at `adr` (through `adr+3` ). The highest byte is stored at `adr` ; the lowest byte is stored at `adr+3` . `adr` must be on a 32-bit boundary; it must be evenly divisible by 4.

**/L**

```
/L  ( -- n )
code# 5c
```

`n` is the size in bytes of a 32-bit word, which is 4.

### /L*

```
/L*  ( n1 -- n2 )
code# 68
```

`n2` is the result of multiplying `n1` by the length in bytes of a (32-bit) long word. This is useful for converting an index into a byte offset. `/L*` is equivalent to `4 *`, but should be used in preference to the less portable `4 *`.

### LA+

```
LA+  ( adr1 index -- adr2 )
code# 60
```

`adr2` is the address of the `index`'th 32-bit longword after `adr1`. For byte-addressed machines (such as this one), this is equivalent to `4 * +`.

`LA+` should be used in preference to the less portable and clear `4 * +`.

### LA1+

```
LA1+  ( adr1 -- adr2 )
code# 64
```

`adr2` is the address of the next 32-bit word after `adr1`. For byte-addressed machines (such as this one), this is equivalent to `4 +`. `LA1+` should be used in preference to the less portable and clear `4 +`.

### LBSPLIT

```
LBSPLIT  ( n -- byte.lo byte byte byte.hi )
code# 7e
```

Splits a 32-bit value into four bytes. The upper bits of each byte are all zeroes.

## LCC

```
LCC  ( char1 -- char2 )
code# 82
```

`char2` is the lower case version of `char1` . If `char1` is not an upper case letter, it is left unchanged. See `UPC` .

## LEAVE

```
LEAVE  ( -- )
generates:  B(LEAVE)
code# 1b
```

Transfers execution to just past the next `LOOP` or `+LOOP` . The loop is terminated and loop control parameters are discarded. May only be used within a `DO` or `?DO` loop.

`LEAVE` may appear within other control structures which are nested within the `DO` loop structure. More than one `LEAVE` may appear within a `DO` loop.

## ?LEAVE

```
?LEAVE  ( flag -- )
generates:  IF LEAVE THEN
generates: B?BRANCH +offset LEAVE B(>RESUME)
code#      14       +offset 1b    b2
```

If flag is `TRUE` (nonzero), transfers control to just beyond the next `LOOP` or `+LOOP` . The loop is terminated and loop control parameters are discarded. If flag is zero, no action is taken. May only be used within a `DO` or `?DO` loop.

`?LEAVE` may appear within other control structures which are nested within the `DO` loop structure. More than one `?LEAVE` may appear within a `DO` loop.

### #LINE

```
#LINE  ( -- adr )
code# 94
```

A VARIABLE that increments whenever CR executes. #LINE @ returns the current value of this VARIABLE . The value in this VARIABLE is used to determine when to pause during long display output, such as DUMP. Its value is reset each time the "ok" prompt displays.

### LINEFEED

```
LINEFEED  ( -- n )
generates:  B(LIT)      10(decimal)
code#       10 00 00 00 0a
```

n is the ASCII code for the linefeed character; decimal 10, hex 0a .

### LOOP

```
LOOP  ( -- )
generates:  B(LOOP) -offset
code#       15      -offset
```

Terminates a DO or ?DO loop. Increments the loop index by one. If the index was incremented across the boundary between limit-1 and limit , the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding DO or ?DO .

For example, a DO loop created as:

```
8 0 DO ... LOOP
```

terminates when the loop index changes from 7 to 8. Thus, the loop will iterate with loop index values from 0 to 7, inclusive.

### +LOOP

```
+LOOP  ( n -- )
generates:  B(+LOOP) -offset
code# 16 -offset
```

Terminates a DO or ?DO loop. Increments the loop index by n (or decrements the index if n is negative). If the index was incremented (or decremented) across the boundary between limit-1 and limit the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding DO or ?DO .

For example, a DO loop created as:

```
8 0 DO ... 2 +LOOP
```

terminates when the loop index crosses the boundary between 7 and 8. Thus, the loop will iterate with loop index values of 0, 2, 4, 6.

By contrast, a `DO` loop created as:

```
0 8 DO ... -2 +LOOP
```

terminates when the loop index crosses the boundary between -1 and 0. Thus, the loop will iterate with loop index values of 8, 6, 4, 2, 0.

## LWSPLIT

```
LWSPLIT  ( n -- word.lo word.hi )
code# 7c
```

Splits the 32-bit value `n` into two 16-bit words. The upper bits of the two generated words are zeroes.

## MAP-SBUS

```
MAP-SBUS  ( physoffset size -- virt )
code# 1 30
```

Creates a memory mapping for some SBus locations, usually within the address space of this SBus card. The MMU page maps are updated, and the generated virtual address is returned.

`physoffset` is the offset into SBus space for the SBus slot which this card is plugged into.

For the SPARCstation 1 and 1+, these values are:

```
Slot#1 - 200.0000
Slot#2 - 400.0000
Slot#3 - 600.0000
```

Note that these values will be different on future architectures. Your FCode program should never use these values explicitly, but should always call `MY-ADDRESS` to get the correct value.

Typical practice is to save the generated virtual address into a `VALUE`.

The memory mapping can (and should) be later undone with `FREE-VIRTUAL`.

Here is a typical usage:

```
( -1 VALUE vregs )
...
MY-ADDRESS  10.0000 +   100
MAP-SBUS   ( virt )
IS vregs
```

## MASK

```
MASK  ( -- adr )
code# 1 24
```

This VARIABLE controls which bits out of every 32-bit longword which will be tested with MEMORY-TEST-SUITE . To test all 32-bits, set MASK to all ones with:

```
ffffffff mask !
```

To test only the low-order byte out of each longword, set just the lower bits of MASK with:

```
000000ff mask !
```

Any arbitrary combination of bits may be tested or masked.

This word is head missing in most boot PROMs.

## MAX

```
MAX  ( n1 n2 -- n3 )
code# 2f
```

n3 is the greater of n1 and n2.

## MEMMAP

```
MEMMAP  ( physoffset space size -- virtual )
code# 1 04
```

Creates a memory mapping for some locations. It updates MMU page maps and returns the generated virtual address. The actual physical address is specified by ( physoffset space ),which indicates the device space and the physical offset within that space.

The memory mapping can (and should) be later undone with FREE-VIRTUAL .

In practice, this word will seldom if ever be used. General system memory is better obtained with ALLOC-MEM or DMA-ALLOC ; and mappings into SBUS space can just as easily be done with MAP-SBUS .

This word is head missing in some early boot PROMs.

## MEMORY-TEST-SUITE

```
MEMORY-TEST-SUITE  ( adr len -- failed? )
code# 1 22
```

Performs a series of tests on some memory, to verify its proper functioning. A TRUE flag is returned if any of the tests failed.

DISPLAY-STATUS is called for each test performed, with a value stored in the variable GROUP-CODE plus the code# for each test.

If DIAGNOSTIC-MODE? is TRUE ( diag-switch? NVRAM parameter is TRUE ), then a message is sent out to the current output device to TTYA if during probe time) giving the name of each test. If any test fails, a "Failed" message will also then be displayed.

For every one of the following tests, the value stored in the variable MASK controls whether only some or all data lines are tested.

For example, to only test data bits 0-23 (skipping bits 24-31), MASK would be set with: `00ffffff MASK !`

The actual tests performed may vary from system to system. On the SPARCstation 1, the tests performed are:

- *Data lines test*. This test performs a walking ones and zeroes on each data line to test for stuck at zero or stuck at one. Code#1.
- *Address quick test*. This tests each address line for being stuck at one, stuck at zero, shorted to another address line, or shorted to a data line. Code#0.
- *Data size test*. Writes a constant 32-bit value to the starting location of the memory, both byte-at-a-time and shortword-

at-a-time, then reads the data back with a 32-bit access and verifies the value. This test verifies proper 8-bit, 16-bit and 32-bit access. Code#4.

The above tests are very fast. If the `diag-switch?` NVRAM parameter is set to `TRUE` , then the following (slower) additional tests are also performed:

- ❏ Data bits test. Tests every bit in memory, by testing a write/read of 0 and a write/read of `ffffffff` at every location. Code#5.
- ❏ Address=data test. Writes each longword location with its own address, then verifies. This checks for the uniqueness of individual locations with RAM chips. Code#2.

## MIN

```
MIN  ( n1 n2 -- n3 )
code# 2e
```

`n3` is the lesser of `n1` and `n2` .

## MOD

```
MOD  ( n1 n2 -- rem )
code# 22
```

`rem` is the remainder after dividing `n1` by the divisor `n2` . `rem` has the same sign as `n2` or is zero. An error condition results if the divisor is zero.

## */MOD

```
*/MOD   ( n1 n2 n3 -- rem quot )
generates:  >R * R> /MOD
code# 30 20 31 2a
```

Calculates `n1 * n2 / n3`, returns the remainder and quotient. The inputs, outputs, and intermediate products are all 32-bit. `rem` has the same sign as `n3` or is zero. An error condition results if the divisor is zero.

## /MOD

```
/MOD  ( n1 n2 -- rem quot )
code# 2a
```

`rem` is the remainder and `quot` is the quotient of `n1` divided by the divsor `n2` . `rem` has the same sign as `n2` or is zero. An error condition results if the divisor is zero.

## MODEL

```
MODEL  ( adr len -- )
code# 1 19
```

This is a shorthand word for creating a `model` property. By convention, `model` identifies the model name/number for an SBus card, for manufacturing purposes. An example usage would be:

```
" SUNW,501-1415-01"  model
```

This is exactly equivalent to the following:

```
" SUNW,501-1415-01" xdrstring  " model"  attribute
```

The `model` property is useful to identify the specific piece of hardware (the SBus card), as opposed to the `name` property (since several different but functionally-equivalent cards would have the same `name` property, thus calling the same SunOS

device driver). Although the `model` property is good to have in general, it generally does not have any other specific purpose.

This word is head missing in most boot PROMs. See also `ATTRIBUTE`.

## MOVE

```
MOVE  ( adr1 adr2 len -- )
code# 78
```

`len` bytes starting at `adr1` (through `adr1+len-1` inclusive) are moved to address `adr2` (through `adr2+len-1` inclusive). If `len` is zero then nothing is moved.

The data are moved such that the `len` bytes left starting at address `adr2` are the same data as was originally starting at address `adr1`. If `adr1` > `adr2` then the first byte of `adr1` is moved first, otherwise the last byte (`len`'th) of `adr1` is moved first. Thus, moves between overlapping fields are properly handled.

`MOVE` will perform 16-bit, 32-bit or possibly even 64-bit operations (for better performance) if the alignment of the operands permits.

## MS

```
MS  ( ms -- )
code# 1 26
```

Delays all execution for the specified number of milliseconds, by executing an empty delay loop for an appropriate number of iterations. The maximum allowable delay will vary from system to system, but is guaranteed to be valid for all values up to at least 1,000,000 (decimal). No other CPU activity takes place during delays invoked using `MS`, although generally this is not a problem for FCode drivers since there is nothing else to do in the meantime anyway. If this word is used excessively, noticeable delays could result.

## MY-ADDRESS

```
MY-ADDRESS  ( -- physoffset )
code# 1 02
```

Returns a physical address, which is the offset into SBus space for the SBus slot that this card is plugged into.

For the SPARCstation 1, these values are:

```
Slot#1 - 200.0000
Slot#2 - 400.0000
Slot#3 - 600.0000
```

Note that these values will be different on future architectures. Your FCode program should never use these values explicitly, but should always call `MY-ADDRESS` to get the correct value.

Typical uses for `MY-ADDRESS` are as a parameter to `MAP-SBUS` or `MEMMAP` , or as part of an attribute declaration (to tell SunOS about various hardware addresses, see "Attributes" especially `REG` and `XDRPHYS` ).

`MY-ADDRESS` is actually a `VALUE`-type variable, which can be changed with IS. The boot PROM automatically sets `MY-ADDRESS` to the correct value before each SBus slot is probed. There should never be any reason for an FCode program to *set* `MY-ADDRESS` to a new value.

Before you can download and execute an FCode program for testing, you must manually set `MY-ADDRESS` to the appropriate value for the slot in which the test SBus device resides.

Also note that `MY-ADDRESS` will change when other slots are probed, so later execution of your routines which use `MY-ADDRESS` could generate illegal results. Sometimes you might need to save `MY-ADDRESS` into a `CONSTANT` during the initial probe, and then always use that saved `CONSTANT` instead.

This approach is not necessary when executing `IS-INSTALL` for a framebuffer, since `MY-ADDRESS` is explicitly restored automatically in this situation.

## MY-PARAMS

```
MY-PARAMS  ( -- adr len )
code# 1 0f
```

`MY-PARAMS` lets you customize information to be passed in an FCode program. The information is in the form of a string (address + length) containing an arbitrary byte sequence.

Typically, the custom information for each SBus device is stored in the system NVRAM, in the `nvram.rc` file area. Just before each SBus slot is probed, the appropriate `MY-PARAMS` information is initialized. Note, however, that this feature is not yet incorporated in current boot PROMs (through 1.3) for the SPARCstation 1. Presently, a null string (`adr` , `len` equals 0, 0) is always passed.

The FCode program retrieves the customizing string by executing `MY-PARAMS` , leaveing the address and length of the string on the stack. The FCode program then interprets the string and performs customizing routines as needed.

The format of the customizing information is completely unspecified, except for consisting of a string of 0 or more bytes. Each SBus card can define whatever format is suited to its needs. For simple Boolean or n-way switches, a single byte could suffice.

For setting the value of various parameters, a simple scheme could consist of sequential pairs of (`keybyte` , `value`) where `keybyte` identifies the parameter to be set, and `value` is an n-byte number whose size depends on the appropriate range of values for that particular parameter.

`value` could also conceivably be a text string for some purpose.

MY-PARAMS is implemented in the boot PROM as a DEFER word as follows:

```
defer my-params
: null-my-params  ( -- adr len )  0 0  ;
' null-my-params  is my-params
```

You can test for proper behavior of MY-PARAMS in an FCode program by first setting an appropriate value for MY-PARAMS (from within the Toolkit) and then manually executing the FCode program.

Here's an example with a text string for a parameter:

```
: new-params  ( -- adr len )  " test-string"  ;
' new-params  is my-params
```

Here's an example with an arbitrary sequence of bytes:

```
create tempname  05 c,  1234ffff l,  73 c,  4321 w,  ff c,
: new-params  ( -- adr len )   tempname  9  ;
' new-params  is my-params
```

This word is head missing in most boot PROMs.

## MY-SPACE

```
MY-SPACE  ( -- space )
code# 1 03
```

Returns a "magic" number, representing the device space which this SBus card is plugged into. For the SPARCstation 1, this will be the same value supplied by the SBUS space Toolkit command (as opposed to OBMEM or OBIO ). Note that this could change on future architectures.

Typical uses for MY-SPACE are as a parameter to MEMMAP or XDRPHYS .

### /N

```
/N  ( -- n )
code# 5d
```

The number of bytes in a normal stack item; 4 in this implementation.

### /N*

```
/N*  ( n1 -- n2 )
code# 69
```

`n2` is the result of multiplying `n1` by the length in bytes of a normal stack item. This is useful for converting an index into a byte offset. This word is equivalent to `4 *` .

### NA+

```
NA+  ( adr1 index -- adr2 )
code# 61
```

`adr2` is the address of the `index`'th "normal" sized word after `adr1` . For this implementation, this is equivalent to `4 * +` .

`NA+` should be used in preference to `WA+` or `LA+` when the intent is to address items that are the same size as items on the stack.

### NA1+

```
NA1+  ( adr1 -- adr2 )
code# 65
```

`adr2` is the address of the next "normal" sized word after `adr1` . For this implementation, this is equivalent to `4 +` or `LA1+` .

`NA1+` should be used in preference to `WA1+` or `LA1+` when the intent is to address items that are the same size as items on the stack.

## NAME

```
NAME  ( adr len -- )
generates: XDRSTRING  " name"            ATTRIBUTE
code#      1 14       12 04 6e 61 6d 65  1 10
```

A shorthand word for creating a name property, used to match an SBus card with the appropriate SunOS driver. The name declaration is required for booting with SunOS, and should be present in every FCode program. For example:

```
" SUNW,bison"  name
```

is exactly equivalent to the following:

```
" SUNW,bison"  xdrstring  " name"  attribute
```

This word is head missing in most boot PROMs. See also `ATTRIBUTE`.

## NAMED-TOKEN

```
NAMED-TOKEN  ( -- )
generates: NAMED-TOKEN string FCode# B(type)
code# b6 len xx xx xx ... 08 xx B(type)
```

`NAMED-TOKEN` (or `NEW-TOKEN` ) is called to create a new dictionary entry. If `HEADERS` are active, `NAMED-TOKEN` is used; if `HEADERLESS` is active, then `NEW-TOKEN` is used.

The new header for a word created with `NAMED-TOKEN` has the following format:

```
NAMED-TOKEN, string, new FCode#, type
```

- The first byte is b6, indicating a `NAMED-TOKEN` format.
- Next is a string containing the name of the new created entry. The string is a length byte and then *length* bytes of text.

- Next is a new FCode# assigned by the tokenizer, starting at `08,00` then `08,01` and working upwards. If `08,ff` is exceeded then `09,00` and so on is used, up to a maximum of `0b,ff`.
- Finally, the `B(`*`type`*`)` byte indicates the type of word being created, such as `B(:)` for colon definition, `B(VALUE)` for `VALUE`s, etc.

`NAMED-TOKEN` should never be used directly in source code.

## NEGATE

```
NEGATE  ( n1 -- n2 )
code# 2c
```

`n2` is the opposite sign of `n1` . This is equivalent to `-1 *` .

## NEW-DEVICE

```
NEW-DEVICE  ( -- )
code# 1 1f
```

Start a new entry in the device tree. This word is used for creating multiple devices in a single FCode program. See `FINISH-DEVICE`. This word is head missing in most boot PROMs.

## NEW-TOKEN

```
NEW-TOKEN  ( -- )
generates: NEW-TOKEN FCode# B(type)
code#      b5 08     xx     B(type)
```

NAMED-TOKEN (or NEW-TOKEN) is called whenever a new dictionary entry is to be created. If HEADERS are active, NAMED-TOKEN is used; if HEADERLESS is active, then NEW-TOKEN is used.

The format for NEW-TOKEN is identical to that for NAMED-TOKEN , except that the string field is missing (and the first byte is b5 instead of b6 ). See NAMED-TOKEN for more details.

NEW-TOKEN should never be used directly in source code.

## NEWLINE

```
NEWLINE  ( -- n )
generates: B(LIT) 10(decimal)
code# 10 00 00 00 0a
```

n is the ASCII code for the character which terminates a line; decimal 10, hex 0a . In this system this is the linefeed character.

## NIP

```
NIP  ( n1 n2 -- n2 )
code# 4d
```

Remove the second item on the stack.

## NOOP

```
NOOP  ( -- )
code# 7b
```

Do nothing. This can be used to waste time or as a placeholder for something that will be patched in later.

## NOT

```
NOT  ( n1 -- n2 )
code# 26
```

`n2` is the one's complement of `n1` , i.e. all the one bits in `n1` are changed to zero, and all the zero bits are changed to one.

This word can also be used to invert a flag, but *only* if any `TRUE` value is a proper `TRUE` : `ffffffff` . Any other non-zero value can also be interpreted as `TRUE` , but `NOT` will *not* turn it into a `FALSE` (zero) value!

## O#

```
O# number  ( -- n )
generates:  B(LIT) value
code# 10 xx xx xx xx
```

Interpret the next number in octal (base 8), regardless of any previous settings of `HEX` , `DECIMAL` , `BINARY`, or `OCTAL` . Only the immediately following number is affected, the default numeric base setting is unchanged. For example:

```
HEX
O# 100  (equals decimal 64)
100     (equals decimal 256)
```

See also `B#`, `D#`, and `H#` .

## OCTAL

```
OCTAL  ( -- )
code# none  -or-
generates: B(LIT)      8  BASE !
code#      10 00 00 00 08 a0   72
```

If outside of a definition, commands the tokenizer program to interpret subsequent numbers in octal (base 8). If within a definition, changes the value in `BASE` , affecting later numeric output when the FCode program is executed. See `BASE`

## OF

```
OF  ( testval -- )
generates:  B(OF) +offset
code# 1c +offset
```

Begin the next test clause in a `CASE` statement. See `CASE` for more details.

## OFF

```
OFF  ( adr -- )
code# 6b
```

Set the 32-bit contents at `adr` to zero ( `FALSE` ).

## OFFSET16

```
OFFSET16  ( -- )
code# cc
```

Instructs the tokenizer program, and the boot PROM, to expect all further branch offsets to be 16-bit values instead of 8-bit values. This allows branches and loops to span a greater range than +-128 bytes; approximately 64-128 source words. Use of this command will result in somewhat larger FCode PROMs.

Sun recommends that you avoid using this command, since well-written Forth programs should not contain definitions larger than 50+ words. It's better to create a larger number of small definitions which build up to the desired function. However, long text strings can consume substantial amounts of space, as can `CASE` statements with many branches.

For downloaded Forth source code, this command is unnecessary, since it only affects the process of conversion to and from the FCode token format. It does not affect the interpretation of typed or downloaded Forth source code.

`OFFSET16` is head missing in many boot PROMs.

## ON

```
ON  ( adr -- )
code# 6a
```

Set the 32-bit contents at `adr` to -1 or `ffff.ffff` ( `TRUE` ).

## OR

```
OR  ( n1 n2 -- n3 )
code# 24
```

`n3` is the bit-by-bit inclusive-or of `n1` with `n2` .

## #OUT

```
#OUT  ( adr -- )
code# 93
```

A `VARIABLE` containing the current column number on the output device. This is updated by `EMIT1` and some other words which modify the cursor position. It is used for display formatting.

## OVER

```
OVER  ( n1 n2 -- n1 n2 n1 )
code# 48
```

The second stack item is copied to the top of the stack.

## 2OVER

```
2OVER  ( n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2 )
code# 54
```

Copies the third and fourth stack items to the stack top.

## PACK

```
PACK  ( adr len pstr -- pstr )
code# 83
```

Convert a byte array (indicated by " `adr len` ") into a packed string, and store it at the location `pstr` . The byte at address `pstr` is the length of the string and the string itself starts at address `pstr+1` .

Packed strings are generally not used in FCode. Virtually all string operations are in the " `adr len` " format.

## >PHYSICAL

```
>PHYSICAL   ( virtual -- physoffset space )
code# 1 06
```

Given a virtual address, return the mapped physical address as a ( `physoffset space` ) pair, specifying the device space (as a "magic number") and the physical offset within that space.

This word has inconsistent behavior in current boot PROMs, and you should avoid using it in FCode programs.

This word is head missing in some early boot PROMs.

## PICK

```
PICK  ( +n -- n2 )
code# 4e
```

n2 is a copy of the +n'th stack value, not counting +n itself. +n must be between 0 and the number of elements on the stack-1 inclusive.

```
0 PICK is equivalent to DUP    ( n1 -- n1 n1 )
1 PICK is equivalent to OVER   ( n1 n2 -- n1 n2 n1 )
2 PICK is equivalent to  ( n1 n2 n3 -- n1 n2 n3 n1 )
```

For readability's sake, the use of PICK should be minimized.

**.R**

```
.R  ( n1 +n -- )
code# 9e
```

n1 is converted using the value of BASE and then displayed right aligned in a field +n characters wide. A leading minus sign is displayed if n is negative. A trailing space is *not* displayed.

If the number of characters required to display n1 is greater than +n, an error condition exists. In this implementation, all the characters required will be displayed, making the resulting field larger than +n.

**R>**

```
R>  ( -- n )
code# 31
```

Removes n from the return stack and places it on the (regular) stack. See >R for restrictions on the use of this word.

**R@**

```
R@  ( -- n )
code# 32
```

`n` is a copy of the top of the return stack. See `>R` for more details.

**>R**

```
>R  ( n -- )
code# 30
```

Removes `n` from the stack and places it on the top of the return stack.

The return stack is a second stack, occasionally useful as a place to temporarily place numeric parameters, i.e. to "get them out of the way" for a little while. However, since the return stack is also used by the system for transferring control from word to word (and by `DO` loops), improper use of `>R` or `R>` is guaranteed to crash your program.

The following restrictions *must* be observed:

- ❏ All values placed on the return stack within a colon definition must be removed before the colon definition is exited, or else the program will crash.
- ❏ Entering a `DO` loop automatically places values onto the return stack. Therefore,
  a) Values placed on the return stack before the loop was started will not be accessible from within the loop.
  b) Values placed on the return stack within the loop must be removed before `LOOP` , `+LOOP` , or `LEAVE` is encountered.
  c) The loop index `I` will no longer be valid when additional values have been placed on the return stack within the loop.

**REG**

```
REG  ( physoffset space size -- )
code# 1 16
```

This is a shorthand word for declaring a property named `reg` (by convention, `reg` is used for declaring the location and size of SBus device registers). Typical usage:

```
my-address  40.0000 +  my-space  20  reg
```

This declares that the device registers are located at offset `40.0000` thru `40.001f` in this slot. The following code would accomplish exactly the same thing:

```
my-address  40.0000 +  my-space  xdrphys
20 xdrint    xdr+
" reg" attribute
```

Note that if you need to declare more than one block of register addresses, you *must* use the longer, more explicit method in order to build the structure to be passed into the `reg` property.

For example, to declare two register fields at 10.0000-10.00ff and 20.0000-20.037f, use the following:

```
my-address  10.0000 +  my-space  xdrphys          \ Offset#1
100 xdrint                              xdr+      \ Merge size#1
my-address  20.0000 +  my-space  xdrphys xdr+     \ Merge offset#2
380 xdrint                              xdr+      \ Merge size#2
" reg" attribute
```

This word is head missing in most boot PROMs. See also `ATTRIBUTE`.

## REPEAT

```
REPEAT  ( -- )
generates:  BBRANCH, -offset, B(>RESOLVE)
code# 13 -offset b2
```

Terminates a `BEGIN ... WHILE ... REPEAT` conditional loop. See `WHILE` for more details.

## ROLL

```
ROLL  ( +n -- )
code# 4f
```

The `+n`'th stack value, not counting `+n` itself, is first removed and then transferred to the top of the stack, moving the remaining values into the vacated position. `+n` must be between 0 and the number of elements on the stack-1, inclusive.

```
0 ROLL is a null operation
1 ROLL is equivalent to SWAP    ( n1 n2 -- n2 n1 )
2 ROLL is equivalent to ROT     ( n1 n2 n3 -- n2 n3 n1 )
3 ROLL is equivalent to         ( n1 n2 n3 n4 -- n2 n3 n4 n1 )
```

For readability's sake, minimize your use of `ROLL`. It is also relatively slow.

## ROT

```
ROT  ( n1 n2 n3 -- n2 n3 n1 )
code# 4a
```

Rotates the top three stack entries, bringing the deepest to the top.

### -ROT

```
-ROT ( n1 n2 n3 -- n3 n1 n2 )
code# 4b
```

Rotates the top three stack entries in the direction opposite from `ROT` , putting the top number underneath the other two.

### 2ROT

```
2ROT  ( n1 n2 n3 n4 n5 n6 -- n3 n4 n5 n6 n1 n2 )
code# 56
```

Rotates the top three pairs of numbers, bringing the third pair to the top of the stack.

### S.

```
S.  ( n -- )
generates:  (.)                          TYPE  BL  EMIT
generates:  DUP ABS <# #S SWAP SIGN #>  TYPE  BL  EMIT
code#       47  2d  96 9a 49   98   97  90    a9  8f
```

Displays the absolute value of `n` in a free-field format with a leading minus sign if `n` is negative. A trailing space is also displayed. Even if the base is hexadecimal, the number will be printed in signed format (see . ) .

### #S

```
#S  ( +L -- 0 )
code# 9a
```

`+L` is converted, appending each resultant character into the pictured numeric output string until the quotient is zero (see: `#` ). A single zero is added to the output string if the number was initially zero. Typically used between `<#` and `#>` . See (.) and (`U.`) for typical usages.

This word is equivalent to calling # repeatedly until the number remaining is zero.

### (S

```
(S text)  ( -- )
code# none
```

Ignore subsequent text after the (S up to a delimiting ) . The same behavior occurs for ( .

Although either (S or \ works equally well for documentation, by common convention we use ( ... ) or (S ... ) for stack comments and \ ... for all other text comments and documentation.

Use (S to distinguish a definition's "interface" stack comment from stack comments *within* a definition (which clarify the current stack state). (This distinction could be of use for implementing automatic stack-checkers.) A code example:

```
\ Map in registers
: map-regs  (s size -- virt )
  reg-addr  swap   ( addr size )
  map-sbus         ( virt )
;
```

### .S

```
.S  ( -- )
code# 9f
```

Displays the contents of the data stack (using . ) in the current base. The top of the stack appears on the right. The contents of the stack are unchanged.

## SBUS-INTR>CPU

```
SBUS-INTR>CPU  ( sbus-intr# -- cpu-intr# )
code# 1 31
```

Convert the SBus interrupt level (1-7) to the CPU interrupt level. The mapping performed will be system-dependent.

This word is called by the `INTR` attribute command.

See `INTR` for a description of the actual mapping performed on the SPARCstation 1.

## SIGN

```
SIGN  ( n -- )
code# 98
```

If `n` is negative, appends an ASCII "-" (minus sign) to the pictured numeric output string. Typically used between `<#` and `#>` . See (.) for a typical usage.

## SPACE

```
SPACE  ( -- )
generates:  BL EMIT
code# a9 8f
```

Display a single space character.

## SPACES

```
SPACES  ( +n -- )
generates:  0 MAX 0  ?DO             SPACE    LOOP
generates:  0 MAX 0  B(?DO) +offset  BL EMIT  B(LOOP) -offset
code#      a5 2f  a5 18     +offset  a9 8f    15      -offset
```

Display `+n` space characters. Nothing is displayed if `+n` is zero.

## SPAN

```
SPAN  ( -- adr )
code# 88
```

A VARIABLE containing the count of characters actually received and stored by the last execution of EXPECT .

**STRUCT**

```
STRUCT  ( -- 0 )
generates:  0
code# a5
```

Initializes a STRUCT ... FIELD structure. See FIELD for details.

**SWAP**

```
SWAP  ( n1 n2 -- n2 n1 )
code# 49
```

Exchanges the top two stack items.

**2SWAP**

```
2SWAP  ( n1 n2 n3 n4 -- n3 n4 n1 n2 )
code# 55
```

Exchanges the top two pairs of stack items.

**THEN**

```
THEN  ( -- )
generates:  B(>RESOLVE)
code# b2
```

Terminate an IF ... THEN or an IF ... ELSE ... THEN conditional structure. See IF for more details.

**TOKENIZER[**

```
TOKENIZER[  ( -- )
code# none
```

This is a tokenizer command, used to cease generating FCode bytes and interpret following text as tokenizer commands (up to the closing `]TOKENIZER`). A `TOKENIZER[ ... ]TOKENIZER` sequence may be used anywhere in an FCode program, either within any definition or outside of definitions.

One plausible use for `TOKENIZER[` would be to generate debugging text during the tokenizing process. (A `CR` flushes the text from the output buffer immediately, which is useful if the tokenizer crashes.) For example:

```
...
TOKENIZER[  .( Step A)  CR  ]TOKENIZER
...
TOKENIZER[  .( Step B)  CR  ]TOKENIZER
...
```

Another use for `TOKENIZER[` is together with `EMIT-BYTE`, to manually output a desired byte of FCode. This would be useful, for example, if you wished to generate a new FCode command which the tokenizer did not understand. For example:

```
...
TOKENIZER[  1 EMIT-BYTE  27 EMIT-BYTE  ]TOKENIZER
\ Manually output FINISH-DEVICE FCode
...
```

## ]TOKENIZER

```
]TOKENIZER  ( -- )
code# none
```

Ends a tokenizer-only command sequence. See `TOKENIZER[`.

## TRUE

```
TRUE  ( -- flag )
```

```
generates:  -1
code# a4
```

Leave the value for the TRUE flag (which is -1) on the stack.

**TUCK**

```
TUCK  ( n1 n2 -- n2 n1 n2 )
code# 4c
```

Copy the top stack item underneath the second item.

**TYPE**

```
TYPE  ( adr len -- )
code# 90
```

A DEFER word which transfers len characters to the output, beginning with the character at address adr , continuing through len consecutive addresses. No action is taken if len is zero.

The output may go either to a framebuffer or to a serial port, depending on which is enabled.

**U.**

```
U.  ( n -- )
code# 9b
```

Display n as an unsigned number in a free-field format, using the current value for BASE . A trailing space is also displayed.

### (U.)

```
(U.)  ( n -- adr len )
generates:  <# #S #>
code# 96 9a 97
```

This is a numeric conversion primitive, used to implement display words such as `U.` . It converts an unsigned number into a string.

### U.R

```
U.R  ( n1 +n -- )
code# 9c
```

`n1` is converted using the value of `BASE` and then displayed as an unsigned number right-aligned in a field `+n` characters wide. A trailing space is *not* displayed.

If the number of characters required to display `n1` is greater than `+n`, an error condition exists. In this implementation, all the characters required will be displayed, making the resulting field larger than `+n`.

### U/MOD

```
U/MOD  ( n1 n2 -- rem quot )
code# 2b
```

`rem` is the remainder and `quot` is the quotient after dividing `n1` by `n2` . All values and arithmetic are unsigned. All values are 32-bit.

### U2/

```
U2/  ( n1 -- n2 )
code# 58
```

`n2` is the result of `n1` logically shifted right one bit. A zero is shifted into the vacated sign bit.

**U<**

```
U<  ( n1 n2 -- flag )
code# 40
```

`flag` is `true` if `n1` is less than `n2` where `n1` and `n2` are treated as unsigned integers.

**U<=**

```
U<=  ( n1 n2 -- flag )
code# 3f
```

`flag` is `true` if `n1` is less than or equal to `n2` where `n1` and `n2` are treated as unsigned integers.

**U>**

```
U>  ( n1 n2 -- flag )
code# 3e
```

`flag` is `true` if `n1` is greater than `n2` where `n1` and `n2` are treated as unsigned integers.

**U>=**

```
U>=  ( n1 n2 -- flag )
code# 41
```

`flag` is true if `n1` is greater than or equal to `n2` where `n1` and n2 are treated as unsigned integers.

## UNTIL

```
UNTIL  ( flag -- )
generates: B?BRANCH -offset
code#      14       -offset
```

Marks the end of a `BEGIN ... ( flag ) UNTIL` conditional loop. When `UNTIL` is encountered, a flag is removed and tested. If the flag is `true`, the loop is terminated and execution continues just after the `UNTIL` . If the flag is false, execution jumps back to just after the corresponding `BEGIN` .

## UPC

```
UPC  ( char1 -- char2 )
code# 81
```

`char2` is the upper case version of `char1` . If `char1` is not a lower case letter, it is left unchanged. See `LCC` .

## VALUE

```
VALUE name  ( n1 -- )   at creation
            ( -- n1 )   at execution
generates:  new header, B(type) = B(VALUE)
code# (header) b8
```

Creates a named, `VALUE`-type variable. The name is initially created with:

```
456 VALUE name
```

where the number before `VALUE` is the initial value for name. Later occurrences of name will leave the correct value on the stack.

You can change the numeric contents of a VALUE variable with IS , as follows:

```
123 IS name
```

VALUE-type variables are widely used in this system.We encourage the use of VALUEs instead of VARIABLEs. VALUEs act similarly to CONSTANTs or colon definitions, in that execution of the word leaves the desired number on the stack. (With a VARIABLE, you always have to do a @ .) This similarity between VALUEs and other words makes the FCode easier to read, write and maintain.

## VARIABLE

```
VARIABLE name  ( -- )       at creation
               ( -- adr )   at execution
generates:  new header, B(type) = B(VALUE)
code# (header) b9
```

Create a named, VARIABLE-type variable. The name is initially created with:

```
VARIABLE name
```

Later occurrences of name leave an address on the stack. The address holds a 32-bit value. To retrieve the value in a VARIABLE and leave it on the stack for subsequent use, enter:

```
name @
```

To change the value in a VARIABLE , enter:

```
123 name !
```

Sun encourages the use of VALUEs instead of VARIABLEs . VALUEs act like CONSTANTs or colon definitions, in that execution of the word leaves the desired number on the stack. (With a VARIABLE , you always have to do a @ .) This similarity between

VALUEs and other words makes the FCode easier to read, write and maintain.

## VERSION

```
VERSION  ( -- n )
code# 87
```

This word is supposed to return a version number saved in the boot PROM. Its actual purpose is poorly-defined and is not consistent within current releases of the boot PROM.

Current values are:

```
SPARCstation 1   PROM 1.0  -  100 (decimal)
                      1.1  -  296 (decimal)
                      1.3  -    0
SPARCstation 1+  PROM 1.3  -    0
```

Sun recommends that you do not use this word in your FCode.

## VERSION1

```
VERSION1  ( -- )
code# fd
```

This byte must be the first byte encountered in the FCode PROM, or else the PROM will be ignored as “non-FCode”. This byte is automatically generated by the FCODE-VERSION1 command.

The word VERSION1 should never be used in FCode source code.

**W!**

```
W! ( n adr -- )
code# 74
```

The low-order 16-bits of `n` are stored at location `adr` (through `adr+1` ). The higher byte is stored at `adr` ; the lower byte is stored at `adr+1` . `adr` must be on a 16-bit boundary; it must be evenly divisible by 2.

**W,**

```
W,  ( n -- )
code# d1
```

Compile 2-bytes into the dictionary. See `C,` for limitations.

**W@**

```
W@  ( adr -- n )
code# 6f
```

Fetch the 16-bit number stored at `adr` (through `adr+1` ). The higher byte is stored at `adr` ; the lower byte is stored at `adr+1` . The remaining high bytes of `n` are set to zero. `adr` must be on a 16-bit boundary; it must be evenly divisible by 2.

**/W**

```
/W  ( -- n )
code# 5b
```

`n` is the size in bytes of a 16-bit word, which is 2.

### /W*

```
/W*  ( n1 -- n2 )
code# 67
```

`n2` is the result of multiplying `n1` by the length in bytes of a (16-bit) word. This is useful for converting an index into a byte offset. `/W*` is equivalent to `2*`, but should be used in preference to `2*` as it is more portable.

### <W@

```
<W@  ( adr -- n )
code# 70
```

Fetches the 16-bit number stored at `adr` (through `adr+1`). The higher byte is stored at `adr`; the lower byte is stored at `adr+1`. The remaining high bytes of `n` are set by sign-extending the upper bit in the higher byte.

### WA+

```
WA+  ( adr1 index -- adr2 )
code# 5f
```

`adr2` is the address of the `index`'th 16-bit word after `adr1`. For byte-addressed machines (such as this one), this is equivalent to `2* +`.

`WA+` should be used in preference to `2* +` because it more clearly expresses the intent of the operation and is more portable.

### WA1+

```
WA1+  ( adr1 -- adr2 )
code# 63
```

`adr2` is the address of the next 16-bit word after `adr1`. For byte-addressed machines (such as this one), this is equivalent to `2+`.

WA1+ should be used in preference to 2+ because it more clearly expresses the intent of the operation and is more portable.

### WBSPLIT

```
WBSPLIT  ( w -- byte.lo byte.next )
code# af
```

Split the two lower bytes of w into two separate bytes (stored as the lower byte of each resulting item on the stack). The upper bytes of w are ignored.

### WFLIP

```
WFLIP  ( n1 -- n2 )
generates:  LWSPLIT SWAP WLJOIN
code# 7c 49 7d
```

Swap the two 16-bit halves of a 32-bit number.

### WHILE

```
WHILE  ( flag -- )
generates:  B?BRANCH +offset
code# 14 +offset
```

Test the exit condition for a BEGIN ... (flag) WHILE ... REPEAT conditional loop. When the WHILE is encountered, a flag is removed from the stack and tested. If the flag is true, execution continues from just after the WHILE through to the REPEAT which then jumps back to just after the BEGIN. If the flag is false, the loop is exited by causing execution to jump ahead to just after the REPEAT.

## WITHIN

```
WITHIN  ( n min max -- flag )
code# 45
```

`flag` is `true` if `n` is between `min` and `max` , inclusive of `min` and exclusive of `max` . ( `min` <= `n` < `max` ) See `BETWEEN` for another version.

## WLJOIN

```
WLJOIN  ( word.lo word.hi -- n )
code# 7d
```

Merge two 16-bit numbers into a 32-bit number.

## XDR+

```
XDR+  ( xdr-adr1 xdr-len1 xdr-adr2 xdr-len2 --
code# 1 12                 -- xdr-adr1 len1+2 )
```

Merge two `xdr`-format strings into a single `xdr`-format string. The two input strings must have been created sequentially. This can be called repeatedly, to create complex, multi-valued `xdr`-format strings for passing to `ATTRIBUTE`.

For example, suppose you wished to create a property named `myprop` with the following information packed sequentially:

```
"size"  2000  "vals"  3  128  40  22
```

This would be coded in FCode as follows:

```
" size"  XDRSTRING
2000     XDRINT     XDR+
" vals"  XDRSTRING  XDR+
3        XDRINT     XDR+
128      XDRINT     XDR+
40       XDRINT     XDR+
22       XDRINT     XDR+
" myprop"  ATTRIBUTE
```

## XDRINT

```
XDRINT  ( n1 -- xdr-adr xdr-len )
code# 1 11
```

Convert an integer into an `xdr`-format string, suitable for passing as a "value" to `ATTRIBUTE` . For example:

```
1152  XDRINT  " hres"  ATTRIBUTE
```

## XDRPHYS

```
XDRPHYS  ( physoffset space -- xdr-adr xdr-len )
code# 1 13
```

Convert a physical address (as a device space and a physical offset) into an `xdr`-format string suitable for `ATTRIBUTE` . For example:

```
MY-ADDRESS  20.0000 +   MY-SPACE  XDRPHYS
" resetloc" ATTRIBUTE
```

## XDRSTRING

```
XDRSTRING  ( adr len -- xdr-adr xdr-len )
code# 1 14
```

Converts an ordinary string, such as created by `"` , into an `xdr`-format string suitable for `ATTRIBUTE` . For example:

```
" MJS,SEH"  XDRSTRING   " authors"  ATTRIBUTE
```

## XOR

```
XOR  ( n1 n2 -- n3 )
code# 25
```

`n3` is the bit-by-bit exclusive-or of `n1` with `n2` .

# A

# *FCode Reference*

## FCode Primitives

The following figures list and briefly describe FCodes currently supported by the Open Boot PROM. Both the FCode token values and Forth names are included.

A token value entry of CR indicates a cross-compiler-generated sequence, while – indicates that no FCode is generated.

*Figure A-1. Stack manipulation.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 51 | DEPTH | (-- +n) | How many items on stack? |
| 46 | DROP | (n --) | Removes n from the stack |
| 52 | 2DROP | (n1 n2 --) | Removes 2 items from stack |
| 47 | DUP | (n -- n n) | Duplicates n |
| 53 | 2DUP | (n1 n2 -- n1 n2 n1 n2) | Duplicates 2 stack items |
| 50 | ?DUP | (n -- n n \| 0) | Duplicates n if it is non-zero |
| CR | 3DUP | (n1 n2 n3 -- n1 n2 n3 n1 n2 n3) | Copies top 3 stack items |
| 4d | NIP | (n1 n2 -- n2) | Discards the second stack item |
| 48 | OVER | (n1 n2 -- n1 n2 n1) | Copy second stack item to top of stack |
| 54 | 2OVER | (n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2) | Copies 2 stack items |
| 4e | PICK | (+n -- n2) | Copies +n-th stack item |
| 30 | >R | (n --) | Moves a stack item to the return stack * |
| 31 | R> | (-- n) | Moves item from return stack to data stack * |
| 32 | R@ | (-- n) | Copies the top of the return stack to the data stack |
| 4f | ROLL | (+n --) | Rotates +n stack items |
| 4a | ROT | (n1 n2 n3 -- n2 n3 n1) | Rotates 3 stack items |
| * Use cautiously. | | | |
| 4b | -ROT | (n1 n2 n3 -- n3 n1 n2) | Shuffles top 3 stack items |

| | | | |
|---|---|---|---|
| 56 | 2ROT | (n1 n2 n3 n4 n5 n6 -- n3 n4 n5 n6 n1 n2) | Rotates 3 pairs of stack items |
| 49 | SWAP | (n1 n2 -- n2 n1) | Exchanges the top 2 stack items |
| 55 | 2SWAP | (n1 n2 n3 n4 -- n3 n4 n1 n2) | Exchanges 2 pairs of stack items |
| 4c | TUCK | (n1 n2 -- n2 n1 n2) | Copies the top stack item below the second item |

*Figure A-2. Arithmetic operations.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 20 | * | (n1 n2 -- n3) | Multiplies n1 times n2 |
| 1e | + | (n1 n2 -- n3) | Adds n1+n2 |
| 1f | - | (n1 n2 -- n3) | Subtracts n1-n2 |
| 21 | / | (n1 n2 -- quot) | Divides n1/n2 |
| CR | 1+ | (n1 -- n2) | Adds one |
| CR | 1- | (n1 -- n2) | Subtracts one |
| 59 | 2* | (n1 -- n2) | Multiplies by 2 |
| 57 | 2/ | (n1 -- n2) | Divides by 2 |
| 27 | << | (n1 +n -- n2) | Left shifts n1 by +n places |
| 28 | >> | (n1 +n -- n2) | Right shifts n1 by +n places |
| CR | <<A | (n1 +n -- n2) | Arithmetic left shift (same as <<) |
| 29 | >>A | (n1 +n -- n2) | Arithmetic right shifts n1 by +n places |
| 2d | ABS | (n -- u) | Absolute value |
| ae | ALIGNED | (adr1 -- adr2) | Adjusts an address to a machine word boundary |
| 23 | AND | (n1 n2 -- n3) | Logical and |
| ac | BOUNDS | (startadr len -- endadr startadr) | Converts start,len to end,start for DO loop |
| 2f | MAX | (n1 n2 -- n3) | n3 is maximum of n1 and n2 |
| 2e | MIN | (n1 n2 -- n3) | n3 is minimum of n1 and n2 |
| 22 | MOD | (n1 n2 -- rem) | Remainder of n1/n2 |
| CR | */MOD | (n1 n2 n3 -- rem quot) | Rem, quotient of n1*n2/n3 |
| 2a | /MOD | (n1 n2 -- rem quot) | Remainder, quotient of n1/n2 |
| 2c | NEGATE | (n1 -- n2) | Changes the sign of n1 |
| 26 | NOT | (n1 -- n2) | One's complement |
| 24 | OR | (n1 n2 -- n3) | Logical or |
| 2b | U/MOD | (ul un -- un.rem un.quot) | Unsigned 32-bit divide of ul/un |
| 58 | U2/ | (u1 -- u2) | Logical right shift 1 bit |
| 25 | XOR | (n1 n2 -- n3) | Exclusive or |

*Figure A-3. Memory operations.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 72 | ! | (n adr --) | Stores a 32-bit number into the variable at adr |
| 6c | +! | (n adr --) | Adds n to the 32-bit number stored in the variable at adr |
| 77 | 2! | (n1 n2 adr --) | Stores 2 numbers at adr; n2 at lower address |
| 76 | 2@ | (adr -- n1 n2) | Fetches 2 numbers from adr; n2 from lower address |
| 6d | @ | (adr -- n) | Fetches a number from the variable at adr |
| CR | ? | (adr16 --) | Display the 32-bit number at adr |
| 75 | C! | (n adr --) | Stores low byte of n at adr |
| 71 | C@ | (adr -- byte) | Fetches a byte from adr |
| CR | BLANK | (adr len --) | Sets len bytes memory to ASCII space at adr |
| CR | CMOVE | (adr1 adr2 u --) | Same as MOVE |
| CR | CMOVE> | (adr1 adr2 u --) | Same as MOVE |
| 7a | COMP | (adr1 adr2 len -- n) | Compares two byte arrays including case. n=0 if same |
| CR | ERASE | (adr len --) | Sets len bytes memory to zero at adr |

| | | | |
|---|---|---|---|
| 79 | FILL | (adr u byte --) | Sets u bytes of memory to byte |
| 73 | L! | (l adr --) | Store the 32-bit number at adr, must be 32-bit aligned |
| 6e | L@ | (adr -- l) | Fetches the 32-bit longword at adr, must be 32-bit aligned |
| 78 | MOVE | (adr1 adr2 u --) | Copies u bytes from adr1 to adr2, handles overlap correctly. |
| 6b | OFF | (adr --) | Stores false (32-bit 0) at adr |
| 6a | ON | (adr --) | Stores true (32-bit -1) at adr |
| 74 | W! | (w adr --) | Stores a 16-bit word at adr, must be 16-bit aligned |
| 6f | W@ | (adr -- w) | Fetches the unsigned 16-bit word at adr, must be 16-bit aligned |
| 70 | <W@ | (adr -- n) | Fetches the signed 16-bit word at adr, must be 16-bit aligned |

*Figure A-4. Comparison operations.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 36 | 0< | (n -- flag) | True if n < 0 |
| 37 | 0<= | (n -- flag) | True if n <= 0 |
| 35 | 0<> | (n -- flag) | True if n <> 0 |
| 34 | 0= | (n -- flag) | True if n = 0, also inverts any flag |
| 38 | 0> | (n -- flag) | True if n > 0 |
| 39 | 0>= | (n -- flag) | True if n >= 0 |
| 3a | < | (n1 n2 -- flag) | True if n1 < n2 |
| 43 | <= | (n1 n2 -- flag) | True if n1 <= n2 |
| 3d | <> | (n1 n2 -- flag) | True if n1 <> n2 |
| 3c | = | (n1 n2 -- flag) | True if n1 = n2 |
| 3b | > | (n1 n2 -- flag) | True if n1 > n2 |
| 42 | >= | (n1 n2 -- flag) | True if n1 >= n2 |
| 44 | BETWEEN | (n min max -- flag) | True if min <= n <= max |
| CR | FALSE | (-- 0) | The value FALSE |
| CR | TRUE | (-- -1) | The value TRUE |
| 40 | U< | (u1 u2 -- flag) | True if u1 < u2, unsigned |
| 3f | U<= | (u1 n2 -- flag) | True if u1 <= u2, unsigned |
| 3e | U> | (u1 n2 -- flag) | True if u1 > u2, unsigned |
| 41 | U>= | (u1 n2 -- flag) | True if u1 >= u2, unsigned |
| 45 | WITHIN | (n min max -- flag) | True if min <= n < max |

*Figure A-5. Text input.*

| dByte | Function | Stack | Description |
|---|---|---|---|
| - | (text) | (--) | Begin a comment (ignored) |
| - | \ | (--) | Ingore rest of line (comment) |
| CR | ASCII x | (-- char) | ASCII value of next character |
| CR | CONTROL x | (-- char) | Interpret next character as ASCII CONTROL character |
| 8e | KEY | (-- char) | Reads a character from the keyboard |
| 8d | KEY? | (-- flag) | True if a key has been typed on the keyboard |
| 8a | EXPECT | (adr +n --) | Gets a line of edited input from the keyboard; store at adr |
| 88 | SPAN | (-- adr) | Variable containing the number of characters read by EXPECT |
| - | (S text) | (--) | Begin a comment (ignored) |

*Figure A-6. ASCII constants.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| ab | BELL | (-- n) | The ASCII code for the bell character; decimal 7 |
| a9 | BL | (-- n) | The ASCII code for the space character; decimal 32 |
| aa | BS | (-- n) | The ASCII code for the backspace character; decimal 8 |
| CR | CARRET | (-- n) | The ASCII code for the carriage return character; decimal 13 |
| CR | LINEFEED | (-- n) | The ASCII code for the linefeed character; decimal 10 |
| CR | NEWLINE | (-- n) | The ASCII code for the newline character; decimal 10 |

*Figure A-7. Numeric input.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| a4 | -1 | (-- n) | Constant -1 |
| a5 | 0 | (-- n) | Constant 0 |
| a6 | 1 | (-- n) | Constant 1 |
| a7 | 2 | (-- n) | Constant 2 |
| a8 | 3 | (-- n) | Constant 3 |
| CR | B# *number* | (-- n) | Interpret next number in binary |
| - | BINARY | (--) | If outside definition, input text in binary |
| CR | D# *number* | (-- n) | Interpret next number in decimal |
| - | DECIMAL | (--) | If outside definition, input text in decimal |
| CR | H# *number* | (-- n) | Interpret next number in hexadecimal |
| - | HEX | (--) | If outside definition, input text in hexadecimal |
| CR | O# *number* | (-- n) | Interpret next number on octal |
| - | OCTAL | (--) | If outside definition, input text in octal |

*Figure A-8. Numeric Primitives.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 99 | # | (+l1 -- +l2) | Converts a digit in pictured numeric output |
| 97 | #> | (l -- adr +n) | Ends pictured numeric output |
| 96 | <# | (--) | Initializes pictured numeric output |
| a0 | BASE | (-- adr) | USER variable containing number base |
| a3 | DIGIT | (char base -- digit true \| char false) | Converts a character to a digit |
| 95 | HOLD | (char --) | Inserts the char in the pictured numeric output string |
| 9a | #S | (+l -- 0) | Converts the rest of the digits in pictured numeric output |
| 98 | SIGN | (n --) | Sets sign of pictured output |

*Figure A-9. Numeric output.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 9d | . | (n --) | Displays a number |
| CR | BINARY | (--) | If inside definition, output in binary |
| CR | .D | (n --) | Display number in decimal |
| CR | DECIMAL | (--) | If intside definition, output in decimal |
| CR | .H | (n --) | Display number in hexadecimal |
| CR | HEX | (--) | If intside definition, output in hexadecimal |
| CR | OCTAL | (--) | If intside definition, output in octal |
| 9e | .R | (n +n --) | Displays a number in a fixed width field |
| 9f | .S | (--) | Displays the contents of the data stack |

| | | | |
|---|---|---|---|
| CR | S. | (n --) | Disaplay n as a signed number |
| 9b | U. | (u --) | Displays an unsigned number |
| 9c | U.R | (u +n --) | Prints an unsigned number in a fixed width field |

*Figure A-10. General -purpose output.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | ." text" | (--) | Compile string for later output |
| CR | .( text) | (--) | Display a string now |
| 91 | (CR | (--) | Output ASCII CR character; hex 0d |
| 92 | CR | (--) | Start a new line of display output |
| 8f | EMIT | (char --) | Displays the character |
| CR | SPACE | (--) | Output a single space character |
| CR | SPACES | (+n --) | Output +n spaces |
| 90 | TYPE | (adr +n --) | Displays n characters |

*Figure A-11. Formatted output.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 94 | #LINE | (-- adr) | Variable holding the line number on the output device |
| 93 | #OUT | (-- adr) | Variable holding the column number on the output device |

*Figure A-12. BEGIN loops.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | AGAIN | (--) | End BEGIN..AGAIN (infinite) loop |
| CR | BEGIN | (--) | Start conditional loop. |
| CR | REPEAT | (--) | Return to loop start |
| CR | UNTIL | (flag --) | If true, exit BEGIN..UNTIL loop |
| CR | WHILE | (flag --) | If true, continue BEGIN..WHILE..REPEAT loop, else exit loop |

*Figure A-13. Conditionals.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | IF | (flag --) | If true, execute next FCode |
| CR | ELSE | (--) | (optional) Execute next FCode if IF failed |
| CR | THEN | (--) | Terminate IF..THEN..ELSE |

*Figure A-14. DO loops.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | DO | (end start --) | Loop, index *start* to *end-1* inclusive |
| CR | ?DO | (end start --) | Like DO, but skip loop if *end = start* |
| 19 | I | (-- n) | Return current loop indix value |
| 1a | J | (-- n) | Return value of next outer loop index |
| CR | LEAVE | (--) | Exit DO loop immediately |
| CR | ?LEAVE | (flag --) | If flag is true, exit DO loop |
| CR | LOOP | (--) | Increment index, return to DO |
| CR | +LOOP | (n --) | Increment by n, return to DO. If n<0, index *start* to *end*. |

*Figure A-15. Control words.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 1d | EXECUTE | (acf --) | Executes the word whose compilation address is on the stack |
| 33 | EXIT | (--) | Returns from the current word |

*Figure A-16. Strings.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | " *text* " | (-- adr len) | Collect a string |
| 84 | COUNT | (pstr -- adr +n) | Unpacks a packed string |
| 82 | LCC | (char -- lower-case-char) | Converts char to lower case |
| 83 | PACK | (adr len pstr -- pstr) | Makes a packed string from adr len, placing it at pstr |
| 81 | UPC | (char -- upper-case-char) | Converts char to upper case |

*Figure A-17. Defining words.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | : (colon) | (--) | Begin colon definition |
| CR | ; (semicolon) | (--) | End colon difinition |
| - | ALIAS *newname oldname* | (--) | Creates *newname* with behavior of *oldname* |
| CR | BUFFER: *name* | (size --) | Create data array of *size* bytes |
| CR | CONSTANT *name* | (n --) | Create a constant |
| CR | CREATE *name* | (--) | Generic defining word |
| CR | DEFER *name* | (--) | Execution vector (change with IS) |
| CR | FIELD *name* | (offset size -- offset+size) | Create a named offset pointer |
| CR | STRUCT | (-- 0) | Initialize for FIELD creation |
| CR | VARIABLE *name* | (--) | Create a data variable |
| CR | VALUE *name* | (n --) | Create named VALUE-type variable (change with IS) |

*Figure A-18. Dictionary compilation.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| d3 | , | (n --) | Places a number in the dictionary |
| d0 | C, | (n --) | Places a byte in the dictionary |
| ad | HERE | (-- adr) | Address of top of dictionary |
| d2 | L, | (l --) | Places a longword in the dictionary |
| d1 | W, | (w --) | Places a word in the dictionary |
| CR | IS name | (n --) | Changes value in a *defer* word or a *value* |

*Figure A-19. Dictionary search.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| CR | ' name | (-- acf) | Find the word (while executing) |
| CR | ['] | (-- acf) | Find word (while compiling) |
| cb | $FIND | (adr len -- adr len false \| acf +-1) | Find a name in the Open PROM |

*Figure A-20. Conversions operators.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 7f | BLJOIN | (b.low b2 b3 b.hi -- l) | Joins four bytes to form a longword |
| b0 | BWJOIN | (b.low b.hi -- w) | Joins two bytes to form a 16-bit word |
| 5a | /C | (-- n) | Address increment for a byte; 1 |
| 66 | /C* | (n1 -- n2) | Multiplies by /C |
| 5e | CA+ | (adr1 index -- adr2) | Increments adr1 by index times /C |
| 62 | CA1+ | (adr1 -- adr2) | Increments adr1 by /C |
| 80 | FLIP | (w1 -- w2) | Swaps the bytes within a 16-bit word |
| 5c | /L | (-- n) | Address increment for a 32-bit longword; 4 |
| 68 | /L* | (n1 -- n2) | Multiplies by /L |
| 60 | LA+ | (adr1 index -- adr2) | Increments adr1 by index times /L |
| 64 | LA1+ | (adr1 -- adr2) | Increments adr1 by /L |
| 7e | LBSPLIT | (l -- b.low b2 b3 b.high) | Split a longword into four bytes |
| 7c | LWSPLIT | (l -- w.low w.high) | Split a longword into two words |
| 5d | /N | (-- n) | Address increment for a normal; 4 |
| 69 | /N* | (n1 -- n2) | Multiplies by /N |
| 61 | NA+ | (adr1 index -- adr2) | Increments adr1 by index times /N |
| 65 | NA1+ | (adr1 -- adr2) | Increments adr1 by /N |
| 5b | /W | (-- n) | Address increment for a 16-bit word; 2 |
| 67 | /W* | (n1 -- n2) | Multiplies by /W |
| 5f | WA+ | (adr1 index -- adr2) | Increments adr1 by index times /W |
| 63 | WA1+ | (adr1 -- adr2) | Increments adr1 by /W |
| af | WBSPLIT | (w -- b.low b.high) | Split a 16-bit word into two bytes |
| CR | WFLIP | (l1 -- l2) | Swap halves of 32-bit longword |
| 7d | WLJOIN | (w.low w.high -- l) | Joins two words to form a longword |

*Figure A-21. Memory buffers allocation.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 8b | ALLOC-MEM | (nbytes -- adr) | Allocates nbytes of memory and returns its address |
| 8c | FREE-MEM | (adr nbytes --) | Frees memory allocated by ALLOC-MEM |

*Figure A-22. Miscellaneous operators.*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 86 | >BODY | (acf -- apf) | Finds parameter field address from compilation address |
| 85 | BODY> | (apf -- acf) | Finds compilation address from parameter field address |
| CR | EMIT-BYTE | (n --) | Output FCode byte (use with TOKENIZER[) |
| 00 | END0 | (--) | Mark the end of Fcode |
| ff | END1 | (--) | Alternate form for END0 (not recommended) |
| CR | FCODE-VERSION1 | (--) | Begin FCode program |
| - | FLOAD *filename* | (--) | Begin tokenizing *filename* |
| - | HEADERLESS | (--) | Create new names with NEW-TOKEN (no name fields) |
| - | HEADERS | (--) | Create new names with NAMED-TOKEN (default) |
| 7b | NOOP | (--) | Does nothing |
| cc | OFFSET16 | (--) | All further branches use 16-bit offsets (instead of 8-bit) |
| - | TOKENIZER[ | (--) | Begin tokenizer program commands |
| - | ]TOKENIZER | (--) | End tokenizer program commands |
| 87 | VERSION | (-- n) | Returns the version# of the Fcode interpreter |

*Figure A-23. Internal operators (invalid for program text).*

| Byte | Function | Stack | Description |
|---|---|---|---|
| 1-f | table#1-f | | Reserved byte codes, used for 2-byte entries |
| 10 | b(lit) | (-- n) | Followed by 32-bit#. Compiled by numeric data |
| 11 | b(') | (-- acf) | Followed by a token (1 or 2-byte code) . Compiled by ['] |
| 12 | b(") | (-- adr len) | Followed by count byte, text. Compiled by " or ." |
| c3 | b(is) | (--) | Compiled by IS |
| fd | version1 | (--) | Followed by null byte, checksum (2 bytes) , length (4 bytes). Compiled by (FCODE-VERSION1) , as the first Fcode bytes |
| fe | 4-byte-id | (--) | Followed by 3 identifier bytes. First Fcode byte, used if no Fcode available, to uniquely identify device |
| 13 | bbranch | (--) | Followed by 8-bit offset. Compiled by ELSE or AGAIN |
| 14 | b?branch | (--) | Followed by 8-bit offset. Compiled by IF or UNTIL |
| 15 | b(loop) | (--) | Followed by 8-bit offset. Compiled by LOOP |
| 16 | b(+loop) | (n --) | Followed by 8-bit offset. Compiled by +LOOP |
| 17 | b(do) | (end start --) | Followed by 8-bit offset. Compiled by DO |
| 18 | b(?do) | (end start --) | Followed by 8-bit offset. Compiled by ?DO |
| 1b | b(leave) | (--) | Compiled by LEAVE or ?LEAVE |
| b1 | b(<mark) | (--) | Compiled by BEGIN |
| b2 | b(>resolve) | (--) | Compiled by ELSE or THEN |
| c4 | b(case) | (--) | Compiled by CASE |
| c5 | b(endcase) | (--) | Compiled by ENDCASE |
| c6 | b(endof) | (--) | Compiled by ENDOF |
| 1c | b(of) | (sel testval -- sel \| none) | Followed by 8-bit offset. Compiled by OF |
| b5 | new-token | (--) | Followed by table#, code#, token-type. ompiled by any defining word. Headerless, not used normally. |
| b6 | named-token | (--) | Followed by packed string (count,text) , table#, code#, token-type Compiled by any defining word (: value constant etc.) |
| b7 | b(:) | | Token-type compiled by : |
| b8 | b(value) | | Token-type compiled by VALUE |
| b9 | b(variable) | | Token-type compiled by VARIABLE |
| ba | b(constant) | | Token-type compiled by CONSTANT |
| bb | b(create) | | Token-type compiled by CREATE |
| bc | b(defer) | | Token-type compiled by DEFER |
| bd | b(buffer:) | | Token-type compiled by BUFFER: |
| be | b(field) | | Token-type compiled by FIELD |
| c2 | b(;) | (--) | End a colon definition. Compiled by ; |

*Figure A-24. Memory allocation.*

| Byte 1 | Byte 2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 01 | DMA-ALLOC | (nbytes -- virt) | Map in nbytes of DMA space, return virt. adr |
| 1 | 02 | MY-ADDRESS | (-- phys) | Return the physical adr of this plug-in device |
| 1 | 03 | MY-SPACE | (-- space) | Return address space of plug-in device "space" is a "magic" number, usable by other routines |
| 1 | 04 | MEMMAP | (phys space nbytes -- virt) | Map in a region, return virtual adrress |
| 1 | 05 | FREE-VIRTUAL | (virt nbytes --) | Free virtual memory from MEMMAP, DMA-ALLOC,or MAP-SBUS |
| 1 | 06 | >PHYSICAL | (virt -- phys space) | Return physical adr and space for virt. adr |

*Figure A-25. Non-volatile parameters.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 0f | MY-PARAMS | (-- adr len) | Return a data array for this plug-in device. The data format is defined specifically for each plug-in device, in order to customize the device. Params for each device, as needed, will be stored in the system NVRAM (not yet implemented) |

*Figure A-26. Device information.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 10 | ATTRIBUTE | (xdr-adr xdr-len name-adr name-len --) | Declare an attribute with the given value structure, for the given name string. |
| 1 | 11 | XDRINT | (n -- xdr-adr xdr-len) | Convert a number into a numeric attribute structure |
| 1 | 12 | XDR+ | (xdr-adr1 xdr-len1 xdr-adr2 xdr-len2 -- xdr-adr1 xdr-len1+2) | Merge two attribute structures. They must have been created sequentially |
| 1 | 13 | XDRPHYS | (phys space -- xdr-adr xdr-len) | Convert physical address and space into an attribute structure |
| 1 | 14 | XDRSTRING | (adr len -- xdr-adr xdr-len) | Convert a string into a value structure |

*Figure A-27. Commmonly-used attributes.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 16 | REG | (phys space size --) | Declare loc. and size of device registers |
| 1 | 17 | INTR | (intr-level vector --) | Declare interrupt vector for this device |
| 1 | 18 | DRIVER | (adr len --) | Declare driver for this device, not recommended |
| 1 | 19 | MODEL | (adr len --) | Declare model# for this device, such as " SUNW,501-1415-01" |
| 1 | 1a | DEVICE-TYPE | (adr len --) | Declare type of device, e.g. " display", " disk", " network", or " byte" |
| CR | | NAME | (adr len --) | Declare SunOS driver name, as in " SUNW,zebra" |

*Figure A-28. Device activation vector setup.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 1c | IS-INSTALL | (acf --) | Identify "install" routine, executes at probe time |
| 1 | 1d | IS-REMOVE | (acf --) | Identify "remove" routine, to deallocate a device |
| 1 | 1e | IS-SELFTEST | (acf --) | Identify "selftest" routine for this device |
| 1 | 1f | NEW-DEVICE | (--) | Open an additional device, using this driver package |
| 1 | 27 | FINISH-DEVICE | (-) | Close out current device, ready for NEW-DEVICE |

*Figure A-29. Self-test utility routines.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 20 | DIAGNOSTIC-MODE? | (-- flag) | Returns "true" if extended diagnostics are desired |
| 1 | 21 | DISPLAY-STATUS | (n --) | Output a selftest status message, with given status# |
| 1 | 22 | MEMORY-TEST-SUITE | (adr len -- status) | Call memory tester for given region |
| 1 | 23 | GROUP-CODE | (-- adr) | Variable, used by MEMORY-TEST-SUITE ((obsolete) |
| 1 | 24 | MASK | (-- adr) | Variable, holds "mask" used by MEMORY-TEST-SUITE |

*Figure A-30. Time utilities.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 25 | GET-MSECS | (-- ms) | Returns the current time, in milliseconds, approx. |
| 1 | 26 | MS | (n --) | Delay for n milliseconds. Resolution is 1 millisecond |

*Figure A-31. Machine-specific support.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 30 | MAP-SBUS | (phys size -- virt) | Map a region of memory in 'sbus' address space |
| 1 | 31 | SBUS-INTR>CPU | (sbus-intr# -- cpu-intr#) | Translate SBus interrupt# into CPU interrupt# |

*Figure A-32. User-set terminal emulation values.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 50 | #LINES | (-- n) | # of lines of text being used for display. This word MUST be initialized (using IS). FBx-INSTALL does this automatically, and also properly icorporates the NVRAM parameter "screen-#rows" |
| 1 | 51 | #COLUMNS | (-- n) | # of columns (chars/line) used for display. This word MUST be initialized (using IS). FBx-INSTALL does this automatically, and also properly incorporates the NVRAM parameter "screen-#columns" |

*Figure A-33. Terminal emulator-set terminal emulation values.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 52 | LINE# | (-- n) | Current cursor position (line#). 0 is top line |
| 1 | 53 | COLUMN# | (-- n) | Current cursor position (column#). 0 is left char. |
| 1 | 54 | INVERSE? | (-- flag) | True if output is inverted (white-on-black) |
| 1 | 55 | INVERSE-SCREEN? | (-- flag) | True if screen has been inverted (black background) |

*Figure A-34. Terminal emulation routines. **

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 57 | DRAW-CHARACTER | (char --) | Paint the given character and advance the cursor |
| 1 | 58 | RESET-SCREEN | (--) | Initialize the display device |
| 1 | 59 | TOGGLE-CURSOR | (--) | Draw or erase the cursor |
| 1 | 5a | ERASE-SCREEN | (--) | Clear all pixels on the display |
| 1 | 5b | BLINK-SCREEN | (--) | Flash the display momentarily |
| 1 | 5c | INVERT-SCREEN | (--) | Change all pixels to the opposite color |
| 1 | 5d | INSERT-CHARACTERS | (n --) | Insert n blanks just before the cursor |
| 1 | 5e | DELETE-CHARACTERS | (n --) | Delete n characters starting at with cursor character, rightward. Remaining chars slide left |
| 1 | 5f | INSERT-LINES | (n --) | Insert n blank lines just before the current line, lower lines are scrolled downward |
| 1 | 60 | DELETE-LINES | (n --) | Delete n lines starting with the current line, lower lines are scrolled upward |
| 1 | 61 | DRAW-LOGO | (line# logoaddr logowidth logoheight --) | Draw the logo. |

* DEFER-type loadable routines.

*Figure A-35. Frame buffer parameter values. **

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 6c | CHAR-HEIGHT | (-- n) | Height (in pixels) of a character (usually 22) |
| 1 | 6d | CHAR-WIDTH | (-- n) | Width (in pixels) of a character (usually 12) |
| 1 | 6f | FONTBYTES | (-- n) | # of bytes/scan line for font entries (usually 2) |
| 1 | 62 | FRAME-BUFFER-ADR | (-- adr) | Address of frame buffer memory |
| 1 | 63 | SCREEN-HEIGHT | (-- n) | Total height of the display (in pixels) |
| 1 | 64 | SCREEN-WIDTH | (-- n) | Total width of the display (in pixels) |
| 1 | 65 | WINDOW-TOP | (-- n) | Distance (in pixels) between display top and text window |
| 1 | 66 | WINDOW-LEFT | (-- n) | Distance (in pixels) between display left edge and text window left edge |

* These must all be initialized before using any FBx- routines.

*Figure A-36. Font operators.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 6a | DEFAULT-FONT | (-- fontbase charwidth charheight fontbytes #firstchar #chars) | Returns default font values, plugs directly into SET-FONT |
| 1 | 6b | SET-FONT | (fontbase charwidth charheight fontbytes #firstchar #chars --) | Set the character font for text output. |
| 1 | 6e | >FONT | (char -- adr) | Return font address for given ASCII character |

*Figure A-37. One-bit framebuffer utilities.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 70 | FB1-DRAW-CHARACTER | (char --) | Paint the character and advance the cursor |
| 1 | 71 | FB1-RESET-SCREEN | (--) | Initialize the display device (noop) |
| 1 | 72 | FB1-TOGGLE-CURSOR | (--) | Draw or erase the cursor |
| 1 | 73 | FB1-ERASE-SCREEN | (--) | Clear all pixels on the display |
| 1 | 74 | FB1-BLINK-SCREEN | (--) | Invert the screen, twice (slow) |
| 1 | 75 | FB1-INVERT-SCREEN | (--) | Change all pixels to the opposite color |
| 1 | 76 | FB1-INSERT-CHARACTERS | (n --) | Insert n blanks just before the cursor |
| 1 | 77 | FB1-DELETE-CHARACTERS | (n --) | Delete n characters, starting at with cursor character, rightward. Remaining chars slide left |
| 1 | 78 | FB1-INSERT-LINES | (n --) | Insert n blank lines just before the current line, lower lines are scrolled downward |
| 1 | 79 | FB1-DELETE-LINES | (n --) | Delete n lines starting with the current line,lower lines are scrolled upward |
| 1 | 7a | FB1-DRAW-LOGO | (line# logoaddr logowidth logoheight --) | Draw the logo. |
| 1 | 7b | FB1-INSTALL | (width height #columns #lines --) | Install the one-bit built-in routines. |
| 1 | 7c | FB1-SLIDE-UP | (n --) | Like FB1-DELETE-LINES, but doesn't clear lines at bottom |

*Figure A-38. Eight-bit framebuffer utilities.*

| Byte1 | Byte2 | Function | Stack | Description |
|---|---|---|---|---|
| 1 | 80 | FB8-DRAW-CHARACTER | (char --) | Paint the character and advance the cursor |
| 1 | 81 | FB8-RESET-SCREEN | (--) | Initialize the display device (noop) |
| 1 | 82 | FB8-TOGGLE-CURSOR | (--) | Draw or erase the cursor |
| 1 | 83 | FB8-ERASE-SCREEN | (--) | Clear all pixels on the display |
| 1 | 84 | FB8-BLINK-SCREEN | (--) | Invert the screen, twice (slow) |
| 1 | 85 | FB8-INVERT-SCREEN | (--) | Change all pixels to the opposite color |
| 1 | 86 | FB8-INSERT-CHARACTERS | (n --) | Insert n blanks just before the cursor |
| 1 | 87 | FB8-DELETE-CHARACTERS | (n --) | Delete n characters starting at with cursor character, rightward. Remaining chars slide left |
| 1 | 88 | FB8-INSERT-LINES | (n --) | Insert n blank lines just before the current line, lower lines are scrolled downward |
| 1 | 89 | FB8-DELETE-LINES | (n --) | Delete n lines starting with the current line, lower lines are scrolled upward |
| 1 | 8a | FB8-DRAW-LOGO | (line# logoaddr logowidth logoheight --) | Draw the logo. |
| 1 | 8b | FB8-INSTALL | (width height #columns #lines --) | Install the eight-bit built-in routines. |

## FCode Byte Values

The following figure lists, in numeric order, currently-assigned FCode byte values.

*Figure A-39. FCode Byte Values.*

| Byte | Name | Stack Comment | |
|---|---|---|---|
| 0 | end0 | (--) | |
| 1 | table1 | | |
| 2 | table2 | | |
| 3 | table3 | | |
| 4 | table4 | | |
| 5 | table5 | | |
| 6 | table6 | | |
| 7 | table7 | | |
| 8 | table8 | | |
| 9 | table9 | | |
| a | table10 | | |
| b | table11 | | |
| c | table12 | | |
| d | table13 | | |
| e | table14 | | |
| f | table15 | | |
| | | | |
| 10 | b(lit) | \ then 32-bit#. | (-- n) |
| 11 | (') | \ then token. | (-- acf) |
| 12 | b(") | \ then cnt,letters. | (-- adr len) |
| 13 | bbranch | \ then offset. | (--) |
| 14 | b?branch | \ then offset. | (--) |
| 15 | b(loop) | \ then offset. | (--) |
| 16 | b(+loop) | \ then offset. | (n --) |
| 17 | b(do) | \ then offset. | (end start --) |
| 18 | b(?do) | \ then offset. | (end start --) |
| 19 | i | (-- index) | |
| 1a | j | (-- outerindex) | |
| 1b | b(leave) | (--) | |
| 1c | b(of) | \ then offset. | (selector testval -- sel \| none) |
| 1d | execute | (acf --) | |
| 1e | + | (n1 n2 -- n3) | |
| 1f | - | (n1 n2 -- n3) | |
| | | | |
| 20 | * | (n1 n2 -- n3) | |
| 21 | / | (n1 n2 -- n3) | |
| 22 | mod | (n1 n2 -- n3) | |
| 23 | and | (n1 n2 -- n3) | |
| 24 | or | (n1 n2 -- n3) | |
| 25 | xor | (n1 n2 -- n3) | |
| 26 | not | (n1 -- n2) | |
| 27 | << | (n1 cnt -- n2) | |
| 28 | >> | (n1 cnt -- n2) | |
| 29 | >>a | (n1 cnt -- n2) | |
| 2a | /mod | (n1 n2 -- rem quot) | |
| 2b | u/mod | (n1 n2 -- rem quot) | |
| 2c | negate | (n1 -- n2) | |

| Byte | Name | Stack Comment |
|---|---|---|
| 2d | abs | (n1 -- n2) |
| 2e | min | (n1 n2 -- n3) |
| 2f | max | (n1 n2 -- n3) |
| | | |
| 30 | >r | (n --) (rs: -- n) |
| 31 | r> | (-- n) (rs: n --) |
| 32 | r@ | (-- n) (rs: --) |
| 33 | exit | (--) |
| 34 | 0= | (n -- flag) |
| 35 | 0<> | (n -- flag) |
| 36 | 0< | (n -- flag) |
| 37 | 0<= | (n -- flag) |
| 38 | 0> | (n -- flag) |
| 39 | 0>= | (n -- flag) |
| 3a | < | (n1 n2 -- flag) |
| 3b | > | (n1 n2 -- flag) |
| 3c | = | (n1 n2 -- flag) |
| 3d | <> | (n1 n2 -- flag) |
| 3e | u> | (n1 n2 -- flag) |
| 3f | u<= | (n1 n2 -- flag) |
| | | |
| 40 | u< | (n1 n2 -- flag) |
| 41 | u>= | (n1 n2 -- flag) |
| 42 | >= | (n1 n2 -- flag) |
| 43 | <= | (n1 n2 -- flag) |
| 44 | between | (n min max -- flag) |
| 45 | within | (n min max -- flag) |
| 46 | drop | (n --) |
| 47 | dup | (n -- n n) |
| 48 | over | (n1 n2 -- n1 n2 n1) |
| 49 | swap | (n1 n2 -- n2 n1) |
| 4a | rot | (n1 n2 n3 -- n2 n3 n1) |
| 4b | -rot | (n1 n2 n3 -- n3 n1 n2) |
| 4c | tuck | (n1 n2 -- n2 n1 n2) |
| 4d | nip | (n1 n2 -- n2) |
| 4e | pick | (+n -- n2) |
| 4f | roll | (+n --) |
| | | |
| 50 | ?dup | (n -- 0 \| n n) |
| 51 | depth | (-- +n) |
| 52 | 2drop | (n1 n2 --) |
| 53 | 2dup | (n1 n2 -- n1 n2 n1 n2) |
| 54 | 2over | (n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2) |
| 55 | 2swap | (n1 n2 n3 n4 -- n3 n4 n1 n2) |
| 56 | 2rot | (n1 n2 n3 n4 n5 n6 -- n3 n4 n5 n6 n1 n2) |
| 57 | 2/ | (n1 -- n2) |
| 58 | u2/ | (n1 -- n2) |
| 59 | 2* | (n1 -- n2) |
| 5a | /c | (-- n) |
| 5b | /w | (-- n) |
| 5c | /l | (-- n) |
| 5d | /n | (-- n) |
| 5e | ca+ | (n1 index -- n2) |

| Byte | Name | Stack Comment |
|---|---|---|
| 5f | wa+ | (n1 index -- n2) |
| | | |
| 60 | la+ | (n1 index -- n2) |
| 61 | na+ | (n1 index -- n2) |
| 62 | ca1+ | (n1 -- n2) |
| 63 | wa1+ | (n1 -- n2) |
| 64 | la1+ | (n1 -- n2) |
| 65 | na1+ | (n1 -- n2) |
| 66 | /c* | (n1 -- n2) |
| 67 | /w* | (n1 -- n2) |
| 68 | /l* | (n1 -- n2) |
| 69 | /n* | (n1 -- n2) |
| 6a | on | (adr --) |
| 6b | off | (adr --) |
| 6c | +! | (n adr --) |
| 6d | @ | (adr -- n) |
| 6e | l@ | (adr -- L) |
| 6f | w@ | (adr -- w) |
| | | |
| 70 | <w@ | (adr -- w) |
| 71 | c@ | (adr -- b) |
| 72 | ! | (n adr --) |
| 73 | l! | (n adr --) |
| 74 | w! | (n adr --) |
| 75 | c! | (n adr --) |
| 76 | 2@ | (adr -- n1 n2) |
| 77 | 2! | (n1 n2 adr --) |
| 78 | move | (adr1 adr2 cnt --) |
| 79 | fill | (adr cnt byte --) |
| 7a | comp | (adr1 adr2 cnt -- n) |
| 7b | noop | (--) |
| 7c | lwsplit | (L -- w.lo w.hi) |
| 7d | wljoin | (w.lo w.hi -- L) |
| 7e | lbsplit | (L -- b.lo b b b.hi) |
| 7f | bljoin | (b.lo b b b.hi -- L) |
| | | |
| 80 | flip | (w1 -- w2) |
| 81 | upc | (char -- upper-case-char) |
| 82 | lcc | (char -- lower-case-char) |
| 83 | pack | (adr len pstr -- pstr) |
| 84 | count | (pstr -- adr len) |
| 85 | body> | (apf -- acf) |
| 86 | >body | (acf -- apf) |
| 87 | version | (-- n) |
| 88 | span | (-- adr) |
| 89 | (reserved) | |
| 8a | expect | (adr +n --) |
| 8b | alloc-mem | (cnt -- adr) |
| 8c | free-mem | (adr cnt --) |
| 8d | key? | (-- flag) |
| 8e | key | (-- char) |
| 8f | emit | (char --) |

| Byte | Name | Stack Comment |
|---|---|---|
| 90 | type | (adr +n --) |
| 91 | (cr | (--) |
| 92 | cr | (--) |
| 93 | #out | (-- adr) |
| 94 | #line | (-- adr) |
| 95 | hold | (char --) |
| 96 | <# | (--) |
| 97 | #> | (L -- adr +n) |
| 98 | sign | (n --) |
| 99 | # | (+L1 -- +L2) |
| 9a | #s | (+L -- 0) |
| 9b | u. | (u --) |
| 9c | u.r | (u cnt --) |
| 9d | . | (n --) |
| 9e | .r | (n cnt --) |
| 9f | .s | (--) |
| | | |
| a0 | base | (-- adr) |
| a1 | (reserved) | |
| a2 | (reserved) | |
| a3 | digit | (char base -- digit true \| char false) |
| a4 | -1 | (-- -1) |
| a5 | 0 | (-- 0) |
| a6 | 1 | (-- 1) |
| a7 | 2 | (-- 2) |
| a8 | 3 | (-- 3) |
| a9 | bl | (-- n) |
| aa | bs | (-- n) |
| ab | bell | (-- n) |
| ac | bounds | (n cnt -- n+cnt n) |
| ad | here | (-- adr) |
| ae | aligned | (adr1 -- adr2) |
| af | wbsplit | (w -- b.lo b.hi) |
| | | |
| b0 | bwjoin | (b.lo b.hi -- w) |
| b1 | b(<mark) | |
| b2 | b(>resolve) | |
| b3 | (reserved) | |
| b4 | (reserved) | |
| b5 | new-token | \ then table#, code#, token-type |
| b6 | named-token | \ then string, table#, code#, token-type |
| b7 | b(:) | \ token-type |
| b8 | b(value) | |
| b9 | b(variable) | |
| ba | b(constant) | |
| bb | b(create) | |
| bc | b(defer) | |
| bd | b(buffer:) | |
| be | b(field) | |
| bf | (reserved) | |
| | | |
| c0 | (reserved) | |

| | Byte | Name | Stack Comment |
|---|---|---|---|
| | c1 | (reserved) | |
| | c2 | b(;) | |
| | c3 | b(is) | |
| | c4 | b(case) | |
| | c5 | b(endcase) | |
| | c6 | b(endof) | |
| | c7- ca | (reserved) | |
| | cb | $find | (adr len -- adr len false \| acf +-1) |
| | cc | offset16 | \ Sets the offset length to 16 bits |
| | cd - cf | (reserved) | |
| | d0 | c, | (n --) |
| | d1 | w, | (n --) |
| | d2 | l, | (n --) |
| | d3 | , | (n --) |
| | d4 - fc | (reserved) | |
| | fd | version1 | \ then 0byte, chksum(2bytes), length(4bytes) |
| | fe | 4-byte-id | \ then 3 more bytes |
| | ff | end1 | |
| 1 | 01 | dma-alloc | (#bytes -- virtual) |
| 1 | 02 | my-address | (-- physical) |
| 1 | 03 | my-space | (-- space) |
| 1 | 04 | memmap | (physical space size -- virtual) |
| 1 | 05 | free-virtual | (virtual len --) |
| 1 | 06 | >physical | (virtual -- physical space) |
| 1 | 0f | my-params | (-- adr len) |
| 1 | 10 | attribute | (xdr-adr xdr-len name-adr name-len --) |
| 1 | 11 | xdrint | (n1 -- xdr-len xdr-len) |
| 1 | 12 | xdr+ | (xdr-adr1 xdr-len1 xdr-adr2 xdr-len2 -- xdr-adr1 xdr-len1+2) |
| 1 | 13 | xdrphys | (paddr space -- xdr-adr xdr-len) |
| 1 | 14 | xdrstring | (adr len -- xdr-adr xdr-len) |
| 1 | 16 | reg | (physical space size --) |
| 1 | 17 | intr | (int-level vector --) |
| 1 | 18 | driver | (adr len --) |
| 1 | 19 | model | (adr len --) |
| 1 | 1a | device-type | (adr len --) |
| 1 | 1c | is-install | (acf --) |
| 1 | 1d | is-remove | (acf --) |
| 1 | 1e | is-selftest | (acf --) |
| 1 | 1f | new-device | (--) |
| 1 | 20 | diagnostic-mode? | (-- flag) |
| 1 | 21 | display-status | (n --) |
| 1 | 22 | memory-test-suite | (adr len -- status) |
| 1 | 23 | group-code | (-- adr) |
| 1 | 24 | mask | (-- adr) |
| 1 | 25 | get-msecs | (-- ms) |
| 1 | 26 | ms | (n --) |
| 1 | 27 | finish-device | (--) |

| | Byte | Name | Stack Comment |
|---|---|---|---|
| 1 | 30 | map-sbus | (phys size -- virt) |
| 1 | 31 | sbus-intr>cpu | (sbus-intr# -- cpu-intr#) |
| | | | |
| 1 | 50 | #lines | (-- n) |
| 1 | 51 | #columns | (-- n) |
| 1 | 52 | line# | (-- n) |
| 1 | 53 | column# | (-- n) |
| 1 | 54 | inverse? | (-- flag) |
| 1 | 55 | inverse-screen? | (-- flag) |
| 1 | 57 | draw-character | (char --) |
| 1 | 58 | reset-screen | (--) |
| 1 | 59 | toggle-cursor | (--) |
| 1 | 5a | erase-screen | (--) |
| 1 | 5b | blink-screen | (--) |
| 1 | 5c | invert-screen | (--) |
| 1 | 5d | insert-characters | (n --) |
| 1 | 5e | delete-characters | (n --) |
| 1 | 5f | insert-lines | (n --) |
| | | | |
| 1 | 60 | delete-lines | (n --) |
| 1 | 61 | draw-logo | (line# laddr lwidth lheight --) |
| 1 | 62 | frame-buffer-adr | (-- addr) |
| 1 | 63 | screen-height | (-- n) |
| 1 | 64 | screen-width | (-- n) |
| 1 | 65 | window-top | (-- n) |
| 1 | 66 | window-left | (-- n) |
| 1 | 6a | default-font | (-- fontbase charwidth charheight fontbytes #firstchar #chars) |
| 1 | 6b | set-font | (fontbase charwidth charheight fontbytes #firstchar #chars --) |
| 1 | 6c | char-height | (-- n) |
| 1 | 6d | char-width | (-- n) |
| 1 | 6e | >font | (char -- adr) |
| 1 | 6f | fontbytes | (-- n) |
| | | | |
| 1 | 70 | fb1-draw-character | (char --) |
| 1 | 71 | fb1-reset-screen | (--) |
| 1 | 72 | fb1-toggle-cursor | (--) |
| 1 | 73 | fb1-erase-screen | (--) |
| 1 | 74 | fb1-blink-screen | (--) |
| 1 | 75 | fb1-invert-screen | (--) |
| 1 | 76 | fb1-insert-characters | (#chars --) |
| 1 | 77 | fb1-delete-characters | (#chars --) |
| 1 | 78 | fb1-insert-lines | (#lines --) |
| 1 | 79 | fb1-delete-lines | (#lines --) |
| 1 | 7a | fb1-draw-logo | (line# logoadr lwidth lheight --) |
| 1 | 7b | fb1-install | (width height #cols #lines --) |
| 1 | 7c | fb1-slide-up | (#lines --) |
| | | | |
| 1 | 80 | fb8-draw-character | (char --) |
| 1 | 81 | fb8-reset-screen | (--) |
| 1 | 82 | fb8-toggle-cursor | (--) |
| 1 | 83 | fb8-erase-screen | (--) |

| | Byte | Name | Stack Comment |
|---|---|---|---|
| 1 | 84 | fb8-blink-screen | (--) |
| 1 | 85 | fb8-invert-scree | (--) |
| 1 | 86 | fb8-insert-characters | (#chars --) |
| 1 | 87 | fb8-delete-characters | (#chars --) |
| 1 | 88 | fb8-insert-lines | (#lines --) |
| 1 | 89 | fb8-delete-lines | (#lines --) |
| 1 | 8a | fb8-draw-logo | (line# ladr lwidth lheight --) |
| 1 | 8b | fb8-install | (width height #cols #lines --) |

# B

# *Framebuffer FCode Reference*

## Introduction

This Appendix describes words which are only useful for creating cards of the *display* DEVICE-TYPE (framebuffers). Your FCode must declare a DEVICE-TYPE of *display* for these words to be applicable.

Words described as head missing do not have the name fields present in some boot PROMs. Thus, although the function can be properly tokenized and executed, it cannot be typed interactively from the Toolkit "ok" prompt, and it cannot be downloaded and executed in source (text) form. The reheader.fth tool will restore the name fields for all such head missing words, so that they may be typed interactively or downloaded in source form.

### BLINK-SCREEN

```
BLINK-SCREEN        ( -- )
code# 1 5b
```

A DEFER word, called by the terminal emulator when needed to flash the entire screen (happens rarely if ever).

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly.

This may be done with IS, or it may be loaded automatically with FB1-INSTALL or FB8-INSTALL (which loads FB1-BLINK-SCREEN or FB8-BLINK-SCREEN, respectively). These default routines invert the screen (twice) by XOR-ing every visible pixel. This is quite slow.

An optional, better replacement routine simply disables the video for 20 milliseconds or so, i.e.

```
: my-blink-screen  ( -- )   video-off  20 ms  video-on  ;
...
(load default behaviors with FBx-INSTALL, then:)
   ['] my-blink-screen  is blink-screen
```

Of course, this example assumes that your display hardware is able to quickly enable and disable the video without otherwise affecting the state.

This word is head missing in most boot PROMs.

## CHAR-HEIGHT

```
CHAR-HEIGHT  ( -- n )
code# 1 6c
```

A VALUE, containing the standard height (in pixels) for all characters to be drawn. This number, when multiplied by #LINES, determines the total height (in pixels) of the active text area.

This word *must* be set to the appropriate value if you wish to use *any* FB1- or FB8- utility routines or >FONT . This may be done with IS, but is normally done by calling SET-FONT .

The standard value for CHAR-HEIGHT (22 decimal) is one of the parameters returned by DEFAULT-FONT .

## CHAR-WIDTH

```
CHAR-WIDTH      ( -- n )
code# 1 6d
```

A VALUE, containing the standard width (in pixels) for all characters to be drawn. This number, when multiplied by #COLUMNS, determines the total width (in pixels) of the active text area.

This word *must* be set to the appropriate value if you wish to use *any* FB1- or FB8- utility routines. This may be done with IS, but is normally done by calling SET-FONT .

The standard value for CHAR-WIDTH (12 decimal) is one of the parameters returned by DEFAULT-FONT .

The FB1 and FB8 character painting support routines in current PROMs do not support widths larger than 16 (decimal). However, it is possible to display wider characters by splitting each character bitmap into 2 halves and calling FBx-DRAW-CHARACTER twice.

## COLUMN#

```
COLUMN#         ( -- n )
code# 1 53
```

A VALUE, set and controlled by the terminal emulator, which contains the current horizontal position of the text cursor. A value of 0 represents the leftmost cursor position (this is *not* the leftmost pixel of the framebuffer - see WINDOW-LEFT ). A value if (typically) 79 decimal represents the rightmost available text position (the actual value is controlled by #COLUMNS ).

This word can (and should) be looked at as needed if your FCode program is implementing its own set of framebuffer primitives.

## #COLUMNS

```
#COLUMNS  ( -- n )
code# 1 51
```

This is a VALUE which returns the number of columns of text, i.e. the number of characters in a line, to be displayed using the boot PROM's terminal emulator. It *must* be set to a proper value in order for the terminal emulator to function correctly.

#COLUMNS is defined in the boot PROM with a reasonable initial value of 80 (decimal), but it should always be actively set by the FCode program. This may be done with IS, or it may be handled automatically as one of the functions performed by FB1-INSTALL or FB8-INSTALL . The value set by FBx-INSTALL or is the smaller of the passed #*cols* parameter and the *screen-#columns* NVRAM parameter.

## DEFAULT-FONT

```
DEFAULT-FONT  ( -- fontbase charwidth
  charheight fontbytes #firstchar #chars )
code# 1 6a
```

This function returns all necessary information about the character font which is built into the boot PROM. This font defines the appearance of every character to be displayed. To load this font, simply pass these parameters to SET-FONT, with: *default-font set-font*

The following discussion describes the font and the passed parameters in more detail. It is not necessary to understand any of this if you are simply using the font as supplied, without any changes. Note that some of the specific information supplied could change in future versions of the CPU boot PROM.

The default font is a gallant.r.19 character font. It is stored in monochrome, i.e. 1 bit per pixel. Since each character is 12 pixels wide and 22 pixels tall, a single font entry contains 22 successive scan line entries (top-to-bottom). Each scan line entry uses 2 bytes to represent 12 pixels. The leftmost pixel is in the MSB of byte#1; the rightmost pixel is in the MSB-3 of byte#2. The 4 LSB's of byte#2 are unused.

Since the top and bottom scan lines of every character are always 0 (blank), we overlap the last (blank) scan line of one

character with the next (blank) scan line of the next character in the font table. Thus, the separation between successive characters in the font table is actually 42 bytes (2*21) instead of the 44 that you would normally expect.

Control characters are unprintable, so the only entries in the font table are for the printable characters (20-7f hex). Any values outside of this range are clipped to the endpoint (20 or 7f), which will both print a "blank".

The actual parameters returned by `DEFAULT-FONT` are:

*fontbase* - The address of the beginning of the built-in font table

*charwidth* - The width of each character in pixels (12 decimal)

*charheight* - The height of each character in pixels (22 decimal)

*fontbytes* - The separation (in bytes) between each scan line entry (2)

*#firstchar* - The ASCII value for the first character actually stored in the font table. This is 20 (hex), for the "blank" character

*#chars* - The total number of characters stored in the font table. This is 60 (hex), i.e. store ASCII characters 20-7f

## DELETE-CHARACTERS

```
DELETE-CHARACTERS  ( n -- )
code# 1 5e
```

A DEFER word, called by the terminal emulator when needed to delete n characters to the right of the cursor. The cursor position is unchanged, and the character under the cursor itself is unchanged, but the first n characters to the right of the cursor are deleted, and all remaining characters to the right of the cursor are moved left by n places. (This command is used during command-line editing.) The end of the line is filled with blanks.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with IS, or it may be loaded automatically with FB1-INSTALL or FB8-INSTALL (which loads FB1-DELETE-CHARACTERS or FB8-DELETE-CHARACTERS , respectively).

This word is head missing in most boot PROMs.

## DELETE-LINES

```
DELETE-LINES  ( n -- )
code# 1 60
```

A DEFER word, called by the terminal emulator to delete n lines starting with the cursor line (and deletes 'n-1' lines below the cursor). Lines above the cursor are unchanged. The cursor position is unchanged. All lines below the deleted lines are scrolled upwards by n lines, and n blank lines are placed at the bottom of the active text area.

This word is used for scrolling, by temporarily moving the cursor to the top of the screen and then calling DELETE-LINES .

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with IS, or it may be loaded automatically with FB1-INSTALL or FB8-INSTALL (which loads FB1-DELETE-LINES or FB8-DELETE-LINES , respectively).

This word is head missing in most boot PROMs.

## DRAW-CHARACTER

```
DRAW-CHARACTER  ( char -- )
code# 1 57
```

A DEFER word, called by the boot PROM's terminal emulator in order to display a single character on the screen at the current cursor location.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with IS, or it may be loaded automatically with FB1-INSTALL or FB8-INSTALL (which loads FB1-DRAW-CHARACTER or FB8-DRAW-CHARACTER , respectively).

This word is head missing in most boot PROMs.

## DRAW-LOGO

```
DRAW-LOGO  ( line# laddr lwidth lheight -- )
code# 1 61
```

A DEFER word, called by the system to display the power-on logo (the graphic displayed on the left side during power-up, or by the BANNER Toolkit command).

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with IS, or it may be loaded automatically with FB1-INSTALL or FB8-INSTALL (which loads FB1-DRAW-LOGO or FB8-DRAW-LOGO , respectively).

It is entirely possible to pack a custom logo into the FCode PROM (for example, as a data array concatenated after the END0 token), and then to redefine DRAW-LOGO to output the custom logo instead, i.e. ignore the passed-in logo and use one of your own instead.

DRAW-LOGO is called by the system using the following parameters:

*line#* - The text line number at which to draw the logo, usually between 0 and 33 (decimal). Generally the LINE#

value is used, to draw the logo at the current cursor position.

*laddr* - The address of the logo template to be drawn. In practice, this will always be either the address of the `OEM-LOGO` field in NVRAM, or the address of a built-in SUN logo. In either case, the logo is a bit array of 64x64 (decimal) pixels (512 bytes). The MSB of the first byte represents the upper-left pixel; MSB-1 represents the next pixel to the right, and so on. A bit value of 1 means that pixel will be painted.

*lwidth* - The width of the passed-in logo (in pixels). Generally this value will always be 64 (decimal).

*lheight*- The height of the passed-in logo (in pixels). Generally this value will always be 64 (decimal).

## ERASE-SCREEN

```
ERASE-SCREEN  ( -- )
code# 1 5a
```

A `DEFER` word, called once during the terminal emulator initialization sequence in order to completely clear all pixels on the display. This word is called just *before* `RESET-SCREEN`, so that the user doesn't actually see the framebuffer data until it has been properly scrubbed.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with `IS`, or it may be loaded automatically with `FB1-INSTALL` or `FB8-INSTALL` (which loads `FB1-ERASE-SCREEN` or `FB8-ERASE-SCREEN`, respectively).

This word is head missing in most boot PROMs.

## FB1-BLINK-SCREEN

```
FB1-BLINK-SCREEN  ( -- )
code# 1 74
```

The built-in default routine to blink or flash the screen momentarily on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the DEFER word BLINK-SCREEN by calling FB1-INSTALL .

This routine is invalid unless the FCode program has called FB1-INSTALL and has initialized FRAME-BUFFER-ADR to a valid virtual address.

This word is implemented simply by calling FB1-INVERT-SCREEN twice. In practice, this can be quite slow (around one full second). It is quite common for a framebuffer FCode program to replace FB1-BLINK-SCREEN with a custom routine which simply disables the video for 20 milliseconds or so, i.e.

```
: my-blink-screen  ( -- )  video-off  20 ms  video-on  ;
...
fb1-install
...
['] my-blink-screen   is blink-screen
```

This word is head missing in most boot PROMs.

## FB1-DELETE-CHARACTERS

```
FB1-DELETE-CHARACTERS  ( n -- )
code# 1 77
```

The built-in default routine to delete n characters to the right of the cursor, on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the DEFER word DELETE-CHARACTERS by calling FB1-INSTALL .

This routine is invalid unless the FCode program has called FB1-INSTALL and SET-FONT and has initialized FRAME-BUFFER-ADR to a valid virtual address.

The cursor position is unchanged, and the character under the cursor is unchanged. The next n characters to the right of the cursor are deleted, and the remaining characters to the right are moved left by n places. The end of the line is filled with blanks.

This word is head missing in most boot PROMs.

### FB1-DELETE-LINES

```
FB1-DELETE-LINES  ( n -- )
code# 1 79
```

The built-in default routine to delete n lines, starting with the cursor line, on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `DEFER` word `DELETE-LINES` by calling `FB1-INSTALL`.

This routine is invalid unless the FCode program has called `FB1-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

The cursor line and `n-1` lines below it are deleted. All lines above the cursor line are unchanged. The cursor position is unchanged. All lines below the deleted lines are scrolled upwards by `n` lines, and `n` blank lines are placed at the bottom of the active text area.

This word is head missing in most boot PROMs.

### FB1-DRAW-CHARACTER

```
FB1-DRAW-CHARACTER  ( char -- )
code# 1 70
```

The built-in default routine for drawing a character on a generic 1-bit-per-pixel framebuffer, at the current cursor location. This routine is loaded into the `DEFER` word `DRAW-CHARACTER` by calling `FB1-INSTALL`.

This routine is invalid unless the FCode program has called `FB1-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

If `INVERSE?` is `TRUE`, then characters are drawn inverted (white-on-black). Otherwise (the normal case) they are drawn black-on-white.

This word is head missing in most boot PROMs.

### FB1-DRAW-LOGO

```
FB1-DRAW-LOGO  ( line# logoadr lwidth lheight -- )
code# 1 7a
```

The built-in default routine to draw the logo on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the DEFER word DRAW-LOGO by calling FB1-INSTALL .

This routine is invalid unless the FCode program has called FB1-INSTALL and SET-FONT and has initialized FRAME-BUFFER-ADR to a valid virtual address.

See DRAW-LOGO for more information on the parameters passed.

This word is head missing in most boot PROMs.

## FB1-ERASE-SCREEN

```
FB1-ERASE-SCREEN  ( -- )
code# 1 73
```

The built-in default routine to clear (erase) every pixel in a generic 1-bit-per-pixel framebuffer. This routine is loaded into the DEFER word ERASE-SCREEN by calling FB1-INSTALL .

This routine is invalid unless the FCode program has called FB1-INSTALL and has initialized FRAME-BUFFER-ADR to a valid virtual address.

All pixels are erased (not just the ones in the active text area). If INVERSE-SCREEN? is TRUE , then all pixels are set to 1, resulting in a black screen. Otherwise (the normal case) all pixels are set to 0, resulting in a white screen.

This word is head missing in most boot PROMs.

## FB1-INSERT-CHARACTERS

```
FB1-INSERT-CHARACTERS  ( n -- )
code# 1 76
```

The built-in default routine to insert n blank characters to the right of the cursor, on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `DEFER` word `INSERT-CHARACTERS` by calling `FB1-INSTALL` .

This routine is invalid unless the FCode program has called `FB1-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

The cursor position is unchanged, but all characters to the right of the cursor are moved right by n places. An error condition exists if an attempt is made to create a line longer than the maximum line size (the value in `#COLUMNS` ).

This word is head missing in most boot PROMs.

## FB1-INSERT-LINES

```
FB1-INSERT-LINES  ( n -- )
code# 1 78
```

The built-in default routine to insert n blank lines below the cursor on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `DEFER` word `INSERT-LINES` by calling `FB1-INSTALL` .

This routine is invalid unless the FCode program has called `FB1-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

The cursor line and all lines above it are unchanged. Any lines pushed off of the bottom of the active text area are lost.

This word is head missing in most boot PROMs.

## CHAR-HEIGHT

```
FB1-INSTALL ( screen-width screen-height
              #cols #lines -- )
code# 1 7b
```

This built-in routine installs all of the built-in default routines for driving a generic 1-bit-per-pixel framebuffer. It also initializes most necessary VALUEs needed for using these default routines.

SET-FONT must be called before FB1-INSTALL is called, because the CHAR-WIDTH and CHAR-HEIGHT values set by SET-FONT are needed when FB1-INSTALL is executed.

FB1-INSTALL loads the following DEFER routines with their corresponding FB1-(whatever) equivalents: RESET-SCREEN , TOGGLE-CURSOR , ERASE-SCREEN , BLINK-SCREEN , INVERT-SCREEN , INSERT-CHARACTERS , DELETE-CHARACTERS, INSERT-LINES , DELETE-LINES , DRAW-CHARACTER , DRAW-LOGO .

The following VALUEs are also initialized:

SCREEN-WIDTH - set to the value of the passed-in parameter "screen-width" (screen width in pixels)

SCREEN-HEIGHT - set to the value of the passed-in parameter "screen-height" (screen height in pixels)

#COLUMNS - set to the smaller of the following two: the passed-in parameter #*cols* , and the NVRAM parameter SCREEN-#COLUMNS

#LINES - set to the smaller of the following two: the passed-in parameter #*lines* , and the NVRAM parameter SCREEN-#LINES

WINDOW-TOP - set to half of the difference between the total screen height ( SCREEN-HEIGHT ) and the height of the active text area ( #LINES times CHAR-HEIGHT )

WINDOW-LEFT - set to half of the difference between the total screen width ( SCREEN-WIDTH ) and the width of the active text area ( #COLUMNS times CHAR-WIDTH ), then rounded down to the nearest multiple of 32 (for performance reasons)

Several internal VALUEs are also set which are used by various FB1- routines.

This word is head missing in most boot PROMs.

### FB1-INVERT-SCREEN

```
FB1-INVERT-SCREEN   ( -- )
code# 1 75
```

The built-in default routine to invert every visible pixel on a generic 1-bit-per-pixel framebuffer. This routine is loaded into the DEFER word INVERT-SCREEN by calling FB1-INSTALL .

This routine is invalid unless the FCode program has called FB1-INSTALL and has initialized FRAME-BUFFER-ADR to a valid virtual address.

All pixels are inverted (not just the ones in the active text area).

This word is head missing in most boot PROMs.

### FB1-RESET-SCREEN

```
FB1-RESET-SCREEN   ( -- )
code# 1 71
```

The built-in default routine to enable a generic 1-bit-per-pixel framebuffer to display data. This routine is loaded into the DEFER word RESET-SCREEN by calling FB1-INSTALL. ( RESET-SCREEN is called just after ERASE-SCREEN during the terminal emulator initialization sequence.)

This word is currently a NOP. Typically, an FCode program will define a hardware-dependent routine to "enable video", and then replace this generic function with:

```
...
 fb1-install
 ...
 ['] my-video-enable  is  reset-screen
```

This word is head missing in most boot PROMs.

### FB1-SLIDE-UP

```
FB1-SLIDE-UP  ( n -- )
code# 1 7c
```

This is a utility routine. It behaves exactly like `FB1-DELETE-LINES`, except that it doesn't clear the lines at the bottom of the active text area. Its only purpose is to scroll the enable plane for framebuffers which have 1-bit overlay and enable planes.

This routine is invalid unless the FCode program has called `FB1-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

This word is head missing in most boot PROMs.

## CHAR-HEIGHT

```
FB1-TOGGLE-CURSOR  ( -- )
code# 1 72
```

The built-in default routine to toggle the cursor location in a generic 1-bit-per-pixel framebuffer. This routine is loaded into the `DEFER` word `TOGGLE-CURSOR` by calling `FB1-INSTALL`. The behavior is to invert every pixel in the one-character-size space for the current position of the text cursor.

This routine is invalid unless the FCode program has called `FB1-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

This word is head missing in most boot PROMs.

## FB8-BLINK-SCREEN

```
FB8-BLINK-SCREEN    ( -- )
code# 1 84
```

The built-in default routine to blink or flash the screen momentarily on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the DEFER word BLINK-SCREEN by calling FB8-INSTALL.

This routine is invalid unless the FCode program has called FB8-INSTALL and has initialized FRAME-BUFFER-ADR to a valid virtual address.

This word is implemented simply by calling FB8-INVERT-SCREEN twice. In practice, this can be very slow (several seconds). It is quite common for a framebuffer FCode program to replace FB8-BLINK-SCREEN with a custom routine which simply disables the video for 20 milliseconds or so, i.e.

```
: my-blink-screen  ( -- )  video-off  20 ms  video-on  ;
...
fb8-install
...
['] my-blink-screen   is blink-screen
```

This word is head missing in most boot PROMs.

## FB8-DELETE-CHARACTERS

```
FB8-DELETE-CHARACTERS  ( n -- )
code# 1 87
```

The built-in default routine to delete n characters to the right of the cursor, on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the DEFER word DELETE-CHARACTERS by calling FB8-INSTALL.

This routine is invalid unless the FCode program has called FB8-INSTALL and SET-FONT and has initialized FRAME-BUFFER-ADR to a valid virtual address.

The cursor position is unchanged, and the character under the cursor is unchanged. The next n characters to the right of the cursor are deleted, and the remaining characters to the right are moved left by n places. The end of the line is filled with blanks.

This word is head missing in most boot PROMs.

### FB8-DELETE-LINES

```
FB8-DELETE-LINES    ( n -- )
code# 1 89
```

The built-in default routine to delete n lines, starting with the cursor line, on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `DEFER` word `DELETE-LINES` by calling `FB8-INSTALL` .

This routine is invalid unless the FCode program has called `FB8-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

The cursor line and `n-1` lines below it are deleted. All lines above the cursor line are unchanged. The cursor position is unchanged. All lines below the deleted lines are scrolled upwards by `n` lines, and `n` blank lines are placed at the bottom of the active text area.

This word is head missing in most boot PROMs.

### FB8-DRAW-CHARACTER

```
FB8-DRAW-CHARACTER   ( char -- )
code# 1 80
```

The built-in default routine for drawing a character on a generic 8-bit-per-pixel framebuffer, at the current cursor location. This routine is loaded into the `DEFER` word `DRAW-CHARACTER` by calling `FB8-INSTALL` .

This routine is invalid unless the FCode program has called `FB8-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

If `INVERSE?` is `TRUE`, then characters are drawn inverted (white-on-black). Otherwise (the normal case) they are drawn black-on-white.

This word is head missing in most boot PROMs.

### FB8-DRAW-LOGO

```
FB8-DRAW-LOGO  ( line# logoadr lwidth lheight -- )
code# 1 8a
```

The built-in default routine to draw the logo on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the DEFER word DRAW-LOGO by calling FB8-INSTALL .

This routine is invalid unless the FCode program has called FB8-INSTALL and SET-FONT and has initialized FRAME-BUFFER-ADR to a valid virtual address.

The logo is drawn by painting every desired pixel with the value 01 (normal characters are painted with the value FF). Typically, color#FF is set to black (for normal black characters), whereas color#01 is set to Sun-blue so that the Sun logo is painted the proper color.

See DRAW-LOGO for more information on the parameters passed.

This word is head missing in most boot PROMs.

## FB8-ERASE-SCREEN

```
FB8-ERASE-SCREEN  ( -- )
code# 1 83
```

The built-in default routine to clear (erase) every pixel in a generic 8-bit-per-pixel framebuffer. This routine is loaded into the DEFER word ERASE-SCREEN by calling FB8-INSTALL .

This routine is invalid unless the FCode program has called FB8-INSTALL and has initialized FRAME-BUFFER-ADR to a valid virtual address.

All pixels are erased (not just the ones in the active text area). If INVERSE-SCREEN? is TRUE, then all pixels are set to FF, resulting in a black screen. Otherwise (the normal case) all pixels are set to 0, resulting in a white screen.

This word is head missing in most boot PROMs.

## FB8-INSERT-CHARACTERS

```
FB8-INSERT-CHARACTERS  ( n -- )
code# 1 86
```

The built-in default routine to insert n blank characters to the right of the cursor, on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `DEFER` word `INSERT-CHARACTERS` by calling `FB8-INSTALL` .

This routine is invalid unless the FCode program has called `FB8-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

The cursor position is unchanged, but all characters to the right of the cursor are moved right by n places. An error condition exists if an attempt is made to create a line longer than the maximum line size (the value in `#COLUMNS` ).

This word is head missing in most boot PROMs.

## FB8-INSERT-LINES

```
FB8-INSERT-LINES  ( n -- )
code# 1 88
```

The built-in default routine to insert n blank lines below the cursor on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the `DEFER` word `INSERT-LINES` by calling `FB8-INSTALL` .

This routine is invalid unless the FCode program has called `FB8-INSTALL` and `SET-FONT` and has initialized `FRAME-BUFFER-ADR` to a valid virtual address.

The cursor line and all lines above it are unchanged. Any lines pushed off of the bottom of the active text area are lost.

This word is head missing in most boot PROMs.

## FB8-INSTALL

```
FB8-INSTALL   ( screen-width screen-height #cols
                #lines -- )
code# 1 8b
```

This built-in routine installs all of the built-in default routines for driving a generic 8-bit-per-pixel framebuffer. It also initializes most necessary VALUEs needed for using these default routines.

SET-FONT must be called before FB8-INSTALL is called, because the CHAR-WIDTH and CHAR-HEIGHT values set by SET-FONT are needed when FB8-INSTALL is executed.

FB8-INSTALL loads the following DEFER routines with their corresponding FB8-(whatever) equivalents: RESET-SCREEN, TOGGLE-CURSOR, ERASE-SCREEN, BLINK-SCREEN, INVERT-SCREEN, INSERT-CHARACTERS, DELETE-CHARACTERS, INSERT-LINES, DELETE-LINES, DRAW-CHARACTER, DRAW-LOGO

The following VALUES are also initialized:

SCREEN-WIDTH - set to the value of the passed-in parameter screen-width (screen width in pixels)

SCREEN-HEIGHT - set to the value of the passed-in parameter screen-height (screen height in pixels)

#COLUMNS - set to the smaller of the following two: the passed-in parameter "#cols", and the NVRAM parameter SCREEN-#COLUMNS

#LINES - set to the smaller of the following two: the passed-in parameter "#lines", and the NVRAM parameter SCREEN-#LINES

WINDOW-TOP - set to half of the difference between the total screen height ( SCREEN-HEIGHT ) and the height of the active text area ( #LINES times CHAR-HEIGHT )

WINDOW-LEFT - set to half of the difference between the total screen width ( SCREEN-WIDTH ) and the width of the active text area ( #COLUMNS times CHAR-WIDTH ), then rounded down to the nearest multiple of 32 (for performance reasons)

Several internal VALUEs are also set which are used by various FB8- routines.

This word is head missing in most boot PROMs.

### FB8-INVERT-SCREEN

```
FB8-INVERT-SCREEN   ( -- )
code# 1 85
```

The built-in default routine to XOR (with hex FF) every visible pixel on a generic 8-bit-per-pixel framebuffer. This routine is loaded into the DEFER word INVERT-SCREEN by calling FB8-INSTALL.

This routine is invalid unless the FCode program has called FB8-INSTALL and has initialized FRAME-BUFFER-ADR to a valid virtual address.

All pixels are inverted (not just the ones in the active text area).

This word is head missing in most boot PROMs.

### FB8-RESET-SCREEN

```
FB8-RESET-SCREEN   ( -- )
code# 1 81
```

The built-in default routine to enable a generic 8-bit-per-pixel framebuffer to display data. This routine is loaded into the DEFER word RESET-SCREEN by calling FB8-INSTALL. ( RESET-SCREEN is called just after ERASE-SCREEN during the terminal emulator initialization sequence.)

This word is currently a NOP. Typically, an FCode program will define a hardware-dependent routine to "enable video", and then replace this generic function with:

```
...
fb8-install
...
['] my-video-enable  is  reset-screen
```

This word is head missing in most boot PROMs.

## FB8-TOGGLE-CURSOR

```
FB8-TOGGLE-CURSOR  ( -- )
code# 1 82
```

The built-in default routine to toggle the cursor location in a generic 8-bit-per-pixel framebuffer. This routine is loaded into the DEFER word TOGGLE-CURSOR by calling FB8-INSTALL . The behavior is to XOR every pixel with FF (hex) in the one-character-size space for the current position of the text cursor.

This routine is invalid unless the FCode program has called FB8-INSTALL and SET-FONT and has initialized FRAME-BUFFER-ADR to a valid virtual address.

This word is head missing in most boot PROMs.

## >FONT

```
>FONT  ( char -- adr )
code# 1 6e
```

This routine converts a character value (ASCII 0-ff) into the address of the font table entry for that character. For the normal, built-in font, only ASCII values 21-7e will result in a printable character, other values will result in pointing to a font entry for "blank".

This word is only of interest if you are implementing your own character-drawing routines. Note that >FONT will generate invalid results unless SET-FONT has been called to initialize the font table to be used.

## FONTBYTES

```
FONTBYTES     ( -- n )
code# 1 6f
```

A VALUE, containing the interval between successive entries in the font table. Each entry contains the next scan line bits for the desired character. Each scan line is normally 12 pixels wide, and is stored as one bit per pixel, thus taking 1 1/2 bytes per scan line. The standard value for FONTBYTES is 2, meaning that the next scan line entry is 2 bytes after the previous one (the last 1/2 byte is wasted space).

This word *must* be set to the appropriate value if you wish to use *any* FB1- or FB8- utility routines or >FONT . This may be done with IS, but is normally done by calling SET-FONT .

The standard value for FONTBYTES (2) is one of the parameters returned by DEFAULT-FONT .

## FRAME-BUFFER-ADR

```
FRAME-BUFFER-ADR   ( -- virt )
code# 1 62
```

This is a VALUE which returns the address of the beginning of framebuffer memory. It *must* be set to an appropriate virtual address (using IS) in order to use *any* of the FB1- or FB8- utility routines. It is suggested that this same VALUE variable be used in any of your custom routines which require a frame buffer address, although of course you are free to create and use your own variable if you wish.

Generally, you should only map in the framebuffer memory just before you are ready to use it, and unmap it if it is no longer needed. Typically, this means you should do your mapping in

your "install" routine, and unmap it in your “remove” routine (see IS-INSTALL and IS-REMOVE ). Here’s some sample code:

```
h# 2.0000  constant  /frame    \ # of bytes in frame buffer
h# 40.0000 constant  foffset   \ Location of frame buffer

: video-map  ( -- )
   my-address  foffset +  /frame  map-sbus  is  frame-buffer-adr
;
: video-unmap  ( -- )
   frame-buffer-adr  /frame  free-virtual
   -1  is  frame-buffer-adr   \ Flag accidental accesses to a
                              \ now-illegal address
;

: power-on-selftest  ( -- )
   video-map
   ( test video memory )
   video-unmap
;
power-on-selftest

: my-install  ( -- )
   video-map
   ...
;
: my-remove   ( -- )
   video-unmap
   ...
;
...
   ['] my-install  is-install
   ['] my-remove   is-remove
```

This word is head missing in most boot PROMs.

## INSERT-CHARACTERS

```
INSERT-CHARACTERS  ( n -- )
code# 1 5d
```

A DEFER word, called by the terminal emulator when needed to insert n blank characters to the right of the cursor. The cursor position is unchanged, but all characters to the right of the cursor are moved right by n places. (This command is used

during command-line editing.) An error condition exists if an attempt is made to create a line longer that the maximum line size (the value in `#COLUMNS` ).

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with `IS`, or it may be loaded automatically with `FB1-INSTALL` or `FB8-INSTALL` (which loads `FB1-INSERT-CHARACTERS` or `FB8-INSERT-CHARACTERS` , respectively).

This word is head missing in most boot PROMs.

## INSERT-LINES

```
INSERT-LINES      ( n -- )
code# 1 5f
```

A `DEFER` word, called by the terminal emulator when needed to insert n blank lines below the cursor. This could be used by a screen editor, for example. The cursor line and all lines above it are unchanged. Any lines "pushed" off of the bottom of the active text area are lost.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with `IS`, or it may be loaded automatically with `FB1-INSTALL` or `FB8-INSTALL` (which loads `FB1-INSERT-LINES` or `FB8-INSERT-LINES` , respectively).

This word is head missing in most boot PROMs.

## INVERSE-SCREEN?

```
INVERSE-SCREEN?  ( -- flag )
code# 1 55
```

A `VALUE`, set and controlled by the terminal emulator, which tells you how to paint the unused portions of each line, i.e. white or black? A value of `TRUE` means paint the unused portion black.

This word can (and should) be looked at as needed if your FCode program is implementing its own set of framebuffer primitives.

This word is head missing in most boot PROMs.

## INVERSE?

```
INVERSE?  ( -- flag )
code# 1 54
```

A VALUE, set and controlled by the terminal emulator, which tells you whether to paint characters as white-on-black or black-on-white. A value of TRUE means white-on-black. Unused characters on each line are not affected (see INVERSE-SCREEN? ).

This word can (and should) be looked at as needed if your FCode program is implementing its own set of framebuffer primitives.

This word is head missing in most boot PROMs.

## INVERT-SCREEN

```
INVERT-SCREEN     ( -- )
code# 1 5c
```

A DEFER word, called by the terminal emulator when needed to invert the entire screen (happens rarely). This routine should XOR every visible pixel.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with IS, or it may be loaded automatically with FB1-INSTALL or FB8-INSTALL (which loads FB1-INVERT-SCREEN or FB8-INVERT-SCREEN, respectively).

This word is head missing in most boot PROMs.

### LINE#

```
LINE#  ( -- n )
code# 1 52
```

A VALUE, set and controlled by the terminal emulator, which contains the current vertical position of the text cursor. A value of 0 represents the topmost line of available text space (this is *not* the topmost pixel of the framebuffer - see WINDOW-TOP ). A typicalvalue of 33 decimal represents the last available line of text space (the actual value is controlled by #LINES ).

This word can (and should) be looked at as needed if your FCode program is implementing its own set of framebuffer primitives.

### #LINES

```
#LINES  ( -- n )
code# 1 50
```

This is a VALUE which returns the number of lines of text to be displayed using the boot PROM's terminal emulator. It *must* be set to a proper value in order for the terminal emulator to function correctly.

#LINES is defined in the boot PROM with a reasonable initial value of 34 (decimal), but it should always be actively set by the FCode program. This may be done with IS, or it may be handled automatically as one of the functions performed by FB1-INSTALL or FB8-INSTALL. The value set by FBx-INSTALL is the smaller of the passed #*lines* parameter and the *screen-#rows* NVRAM parameter.

### RESET-SCREEN

```
RESET-SCREEN  ( -- )
code# 1 58
```

A DEFER word, called by the boot PROM's terminal emulator (just after ERASE-SCREEN ). This word is called only once, during the terminal emulator initialization sequence, in order to enable the framebuffer to display information. A typical use for this function is to "enable video".

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with IS, or it may be loaded automatically with FB1-INSTALL or FB8-INSTALL (which loads FB1-RESET-SCREEN or FB8-RESET-SCREEN , respectively). These words are NOPs, so it is very common to first call FBx-INSTALL and then to override the default setting for RESET-SCREEN with:

```
['] my-video-on   is reset-screen
```

This word is head missing in most boot PROMs.

## SCREEN-HEIGHT

```
SCREEN-HEIGHT   ( -- n )
code# 1 63
```

A VALUE, containing the height of the display (in pixels). It may also be interpreted as the number of "lines" of memory.

This word is initially set to 900 (decimal), but should always be set explicitly to the appropriate value if you wish to use the FB1- or FB8- utility routines. This may be done with IS, or it may be loaded as one of the parameters to FB1-INSTALL or FB8-INSTALL .

In particular, this value is used in FBx-INVERT / ERASE / BLINK-SCREEN and in calculating WINDOW-TOP .

Typical code might create a CONSTANT called *vres* . This would be used as the *height* parameter for FBx-INSTALL , and might also be passed as an ATTRIBUTE to SunOS if needed.

This word is head missing in most boot PROMs.

## SCREEN-WIDTH

```
SCREEN-WIDTH  ( -- n )
code# 1 64
```

A VALUE, containing the width of the display (in pixels). It may also be interpreted as the number of pixels (in memory) between one screen location and the next location immediately below it. The latter definition takes precedence if there is a conflict (e.g. there are unused/invisible memory locations at the end of each line).

Typical code might create a CONSTANT called *hres* . This would be used as the *width* parameter for FBx-INSTALL , and might also be passed as an ATTRIBUTE to SunOS if needed.

This word is head missing in most boot PROMs.

## SET-FONT

```
SET-FONT    ( fontbase charwidth charheight
              fontbytes #firstchar #chars --)
code# 1 6b
```

This routine declares the font table to be used for printing characters on the screen. This routine *must* be called if you wish to use *any* of the FB1- or FB8- utility routines or >FONT .

Normally, SET-FONT is called just after DEFAULT-FONT . DEFAULT-FONT leaves on the stack the exact set of parameters needed by SET-FONT . This approach allows your FCode program to inspect and/or alter the default parameters if desired. See DEFAULT-FONT for more information on these parameters.

## TOGGLE-CURSOR

```
TOGGLE-CURSOR  ( -- )
code# 1 59
```

A DEFER word, called by the boot PROM's terminal emulator just before any character or string is printed, and just afterwards as well. (It is also called once during the terminal emulator initialization sequence.) The normal behavior of this word is to XOR the pixels at the current cursor position, in order to leave a colored rectangle marking the next character to be output.

This word is initially empty, but *must* be loaded with an appropriate routine in order for the terminal emulator to function correctly. This may be done with `IS`, or it may be loaded automatically with `FB1-INSTALL` or `FB8-INSTALL` (which loads `FB1-TOGGLE-CURSOR` or `FB8-TOGGLE-CURSOR` , respectively).

This is a good place to perform any necessary "cleanup" of display hardware state, such as resetting color maps or selecting the proper modes. For example, a window system may have set a color lookup table so that the color used for displaying text does not contrast with the background. If the PROM terminal emulator is then asked to display some system messages on the screen, the messages would be unreadable. Consequently, it would be a good idea to restore the text entries in the color lookup table in the `TOGGLE-CURSOR` routine.

This word is head missing in most boot PROMs.

## WINDOW-LEFT

```
WINDOW-LEFT  ( -- n )
code# 1 66
```

A VALUE, containing the offset (in pixels) of the left edge of the active text area from the left edge of the visible display. The "active text area" is where characters are actually printed. (There is generally a border of unused blank area surrounding it on all sides.) `WINDOW-LEFT` contains the size of the left portion of the unused border.

The size of the right portion of the unused border is determined by the difference between `SCREEN-WIDTH` and the sum of `WINDOW-LEFT` plus the width of the active text area ( `#COLUMNS` times `CHAR-WIDTH` ).

This word is initially set to 0, but should always be set set explicitly to the appropriate value if you wish to use *any* `FB1-` or `FB8-` utility routines. This may be done with `IS`, or it may be set automatically by calling `FB1-INSTALL` or `FB8-INSTALL` .

When set with `FBx-INSTALL` , a calculation is done to set `WINDOW-LEFT` so that the available unused border area is evenly split between the left border and the bottom border. (The calculated value for `WINDOW-LEFT` is rounded down to the

nearest multiple of 32, though. This allows all pixel-drawing to proceed more efficiently.) If you wish to use FBx-INSTALL but desire a different value for WINDOW-TOP , simply change it with IS *after* calling FBx-INSTALL .

This word is head missing in most boot PROMs.

## WINDOW-TOP

```
WINDOW-TOP      ( -- n )
code# 1 65
```

A VALUE, containing the offset (in pixels) of the top of the active text area from the top of the visible display. The "active text area" is where characters are actually printed. (There is generally a border of unused blank area surrounding it on all sides.) WINDOW-TOP contains the size of the top portion of the unused border.

The size of the bottom portion of the unused border is determined by the difference between SCREEN-HEIGHT and the sum of WINDOW-TOP plus the height of the active text area ( #LINES times CHAR-HEIGHT ).

This word is initially set to 0, but should always be set set explicitly to the appropriate value if you wish to use *any* FB1- or FB8- utility routines. This may be done with IS, or it may be set automatically by calling FB1-INSTALL or FB8-INSTALL . When set with FBx-INSTALL , a calculation is done to set WINDOW-TOP so that the available unused border area is evenly split between the top border and the bottom border. If you wish to use FBx-INSTALL but desire a different value for WINDOW-TOP , simply change it with IS *after* calling FBx-INSTALL .

This word is head missing in most boot PROMs.

# C

# *Sample FCode Driver*

## Framebuffer Driver

The following FCode program is an example of a typical framebuffer driver.

```
\   Example FCode driver for a fictitious "model xyz" 1-bit frame
\   buffer

\   This is the source code for an FCode driver for a fictitious frame
\   buffer device.  This is intended as an example of how to write an
\   FCode driver.  We assume the following hardware characteristics:
\
\   The device is an SBus card manufactured by "XYZ Industries,
\   Inc.", whose stock is traded under the symbol "XYZI".
\
\   The UNIX driver for the device will be called "/dev/bwxyz"
\
\   The device is a 1-bit-deep framebuffer, with one bit per pixel.
\   The leftmost pixel corresponds to the most significant bit of a
\   32-bit word, the following pixel corresponds to the following most
\   significant bit, and so on.
\
\   The displayed resolution is 1280 pixels in the "x" (horizontal)
\   direction, and 1024 pixels in the "y" (vertical) direction.
\   However, successive vertical lines are separated by a (2048 / 8)
\   addresses. In other words, there are (2048 - 1280) / 8 memory
\   locations on each line that do not correspond to visible pixels.
```

```
\
\   The framebuffer memory begins at offset (hex) 80.0000 from the start
\   of of the card's address space.
\
\   There is a 32-bit control/status register at offset (hex) 40.0000.
\   The low order bit on that register turns the video on and off.
\   When that bit is 0, the screen is dark.  When that bit is 1, the
\   screen displays the contents of the framebuffer.
\
\   Since this framebuffer is similar to the "generic" 1-bit frame
\   buffer for which the Boot PROM already has graphics routines, we
\   will mostly use those generic routines, substituting our own
\   routines in a few places.
\
\   To enhance readablity, all pre-defined FCodes are in upper case, and
\   newly-defined FCodes are in lower case.  (All FCode commands are
\   still case-insensitive.)

\   FCODE-VERSION1 identifies the language in which this program is
\   written.
\   The FCode compiler generates a header that is recognized by the Boot
\   PROM, causing the Boot PROM to proceed with the execution of this
\   program.

FCODE-VERSION1

HEX                              \ Select the default number base

\   The following declarations identify the device.  Associated with each
\   device is a "device node".  Device nodes are maintained by
\   the Boot PROM, and UNIX can look at them.  Each device node has a
\   property list containing an arbitrary number (name,value) pairs.
\   Names and values are both strings.
\

" XYZI,bwxyz"    NAME            \ Name of UNIX driver
" XYZI,12345-01" MODEL          \ Manufacturer's model number
" display"       DEVICE-TYPE    \ Device class, used by Boot PROM

\   The NAME command declares the value of the " name" property,
\   which is the name of the UNIX driver that should be used for this
\   device.
\   This is a hint to the autoconfiguration code in UNIX, helping it to
\   select an appropriate driver.
\
```

```
\   The driver name includes the name of the manufacturer, so that
\   driver names will not clash between different manufacturers.
\   The suggested form of the manufacturer name is the manufacturer's
\   stock symbol, being reasonably unique and well-known.
\
\   Companies that do not have a stock symbol can use any other name
\   they wish, so long as it does not clash with stock symbols.
\
\   The MODEL number declares the value of the existing " model"
\   property, which is the manufacturer's part number.  It should
\   start with the manufacturer's name, as in the driver name.  The
\   rest of the string can be whatever is appropriate for a given
\   manufacturer.
\
\   The DEVICE-TYPE string identifies the primary use of the device,
\   as the " device_type" property.
\   If the device type is one that the Boot PROM knows, the Boot PROM
\   may attempt to use that device during the startup/boot phase.
\   Currently, the Boot PROM knows about the device types " display",
\   " disk", " tape", and " network".  Other device types not in this
\   list are ignored by the Boot PROM.  In either case, the device type
\   string is added to the property list.

\   You can otionally add a source code control ID or a copyright notice.
\   This is not necessary, and is not used, as far as the execution of
\   the software is concerned.

: sccsid  ( -- adr len )  " xyzi.bw.fth  1.3  90/08/08"  ;

: copyright  ( -- adr len )
   " Copyright (c) 1989 by XYZ Industries, Inc. "
;
\   Define constants for some hardware parameters
\   Note that the values for  mem-bytes  and  frame-buf-size  are
\   calculated from other parameters.  The calculation is performed once,
\   at the time that the Boot PROM first interprets the FCode PROM.

D# 2048 CONSTANT mempixels      \ Pixels per line in the memory
mempixels 8 / CONSTANT membytes \ Bytes per scan line

D# 1280 CONSTANT hres           \ Horizontal resolution ( pixels/line)
D# 1024 CONSTANT vres           \ Vertical resolution ( lines/screen)

membytes vres *                 \ Calculate size of framebuffer memory
   CONSTANT frame-buf-size      \ Size of framebuffer
```

```
\   The following VALUE-type variables are initialized to -1 .  Their
\   values will later be set to dynamically-assigned virtual addresses.

-1 VALUE status-register        \ status register virtual address var.

\   Now we define some procedures.
\
\   xyz-map  establishes a virtual-to-physical mapping for each of the
\   addressable regions on the board (except for the FCode PROM, which is
\   handled automatically by the Boot PROM).  MY-ADDRESS is a predefined
\   procedure that returns the physical address of the slot in which
\   this board is located (as determined by the Boot PROM).

: xyz-map  ( -- )

\  Base-address Offset   Size  create-mapping   remember-virtual-address

   MY-ADDRESS  40.0000 +   4              MAP-SBUS    IS status-register
   MY-ADDRESS  80.0000 +   frame-buf-size MAP-SBUS    IS FRAME-BUFFER-ADR
;

\   xyz-unmap destroys the mappings previous created by xyz-map ,
\   letting their virtual memory resources be reused by other devices.

: xyz-unmap  ( -- )

\    Virtual-address   Size            give-it-back

   status-register   4               FREE-VIRTUAL
   FRAME-BUFFER-ADR  frame-buf-size  FREE-VIRTUAL
;


\   Procedures to turn the video on and off using the video enable bit
\   in the status register

: video-off  ( -- )
   0 status-register l!
;

: video-on   ( -- )
   1 status-register l!
;

\   The "generic" blink-screen routine is very slow, since it has to
```

```
\   invert all the pixels on the screen twice.  We take advantage of
\   the device's ability to turn the video on and off, in order to
\   blink the screen for a reasonable interval.  20 milliseconds
\   seems to be about right.

: xyz-blink-screen  ( -- )
   video-off   D#  20 MS   video-on
;

\   Installation and removal procedures that are called by the Boot PROM
\   if this device is selected for use during  booting/startup process.


\   The install procedure is called by the Boot PROM only if this device
\   is selected for use as the startup display device (console).  Install
\   procedure prepares the device for such use, and exports the interface
\   procedures that implement the device's behavior as a display device.

: xyz-install  ( -- )
   xyz-map                      \   Establish virtual mappings

   DEFAULT-FONT SET-FONT        \   Select the default font

   \   Export the virtual address of the framebuffer, so that it can
   \   be re-used by the driver.  Large framebuffers consume lots of
   \   virtual memory, so we don't want to have to have it mapped twice.

   FRAME-BUFFER-ADR XDRINT    " address"   ATTRIBUTE

   \   Install "generic" 1-bit framebuffer routines, with appropriate
   \   resolution parameters.

   \   bytes/line  lines    max-chars/line      max-lines

   membytes  vres   hres CHAR-WIDTH  /   vres CHAR-HEIGHT /   FB1-INSTALL
```

```
   \   Now we substitute our own procedures in place of some "generic"
   \   procedures that are unsuitable for our hardware

   ['] xyz-blink-screen       IS BLINK-SCREEN
   ['] video-on               IS RESET-SCREEN
;

\   The remove procedure is called by the Boot PROM if the device is no
\   longer to be used as the display device.  This remove procedure turns
\   off the device and frees up any resources that it consumes.

: xyz-remove  ( -- )
   video-off  xyz-unmap
;

\   The selftest procedure is called by the Boot PROM if the user has
\   requested that the device be tested.

: xyz-selftest  ( -- status )
   xyz-map

   DIAGNOSTIC-MODE?  IF  ." Testing Sbus framebuffer" CR  THEN

   H# ffffffff MASK !
   FRAME-BUFFER-ADR  frame-buf-size  MEMORY-TEST-SUITE  ( status )

   xyz-unmap
;
```

```
\   Now for the final procedure.  The following code will
\   execute once, when the Boot PROM first interprets the FCode PROM.

\   Clear the device, in case it is not automatically initialized
\   at power-up time.

: xyz-probe ( -- )

xyz-map  video-off  xyz-unmap   \     Turn off the screen


\   Export some more properties used by the UNIX auto-configuration code
\
\   Note the use of "xdr" construction procedures in the creation of the
\   property value for the " reg" property.  These procedures encode
\   various data types into a byte array, in a machine-independent
\   format.

MY-ADDRESS 40.0000 +  MY-SPACE  XDRPHYS     \ address of status register
4               XDRINT  XDR+                \ size of status register
MY-ADDRESS 80.0000 + MY-SPACE XDRPHYS  XDR+ \ address of framebuffer
frame-buf-size  XDRINT  XDR+                \ size of framebuffer
   " reg" ATTRIBUTE

5  0  INTR   \     Interrupt on SBus level 5 with null vector


\   The following properties that might be handy to pass up to the UNIX
\   driver, in case it is written to handle framebuffers with differing
\   resolutions.  Many of the Sun framebuffer drivers are parameterized
\   in this way.  If the UNIX driver for this device does not need these
\   properties, they may be omitted.

1          XDRINT   " depth"        ATTRIBUTE
membytes   XDRINT   " linebytes"    ATTRIBUTE
hres       XDRINT   " width"        ATTRIBUTE
vres       XDRINT   " height"       ATTRIBUTE
```

```
\   Export the standard interface procedures that let the Boot PROM
\   use this device driver.

['] xyz-install   IS-INSTALL
['] xyz-remove    IS-REMOVE
['] xyz-selftest  IS-SELFTEST
;

xyz-probe                       \ Now, execute the new procedure.


\   This is the end of the FCode program

END0
```

# *Index*

## E



## F



## H



## I

## M



## N



## O



## P



## R



## S



## T



## U

## V



## W